Lecture Notes in Computer Science 1405

Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Suzanne M. Embury Nicholas J. Fiddian
W. Alex Gray Andrew C. Jones (Eds.)

Advances in Databases

16th British National Conference on Databases
BNCOD 16
Cardiff, Wales, UK, July 6-8, 1998
Proceedings

Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Suzanne M. Embury
Nicholas J. Fiddian
W. Alex Gray
Andrew C. Jones
Cardiff University, Department of Computer Science
P.O. Box 916, Cardiff CF2 3XF, UK
E-mail: {S.M.Embury, N.J.Fiddian, W.A.Gray, Andrew.C.Jones}@cs.cf.ac.uk

Cataloging-in-Publication data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Advances in databases : proceedings / 16th British National Conference on Databases, BNCOD 16, Cardiff, Wales, UK, July 6 - 8, 1998. Suzanne M. Embury ... (ed.). - Berlin ; Heidelberg ; New York ; Barcelona ; Budapest ; Hong Kong ; London ; Milan ; Paris ; Santa Clara ; Singapore ; Tokyo : Springer, 1998
(Lecture notes in computer science ; Vol. 1405)
ISBN 3-540-64659-0

CR Subject Classification (1991): H.2, H.3, H.4

ISSN 0302-9743
ISBN 3-540-64659-0 Springer-Verlag Berlin Heidelberg New York

Typesetting: Camera-ready by author
SPIN 10637142 06/3142 - 5 4 3 2 1 0 Printed on acid-free paper

Foreword

This volume contains the proceedings of the 16th British National Conference on Databases (BNCOD-16) held at Cardiff University in July 1998.

The aim of BNCOD is to present new developments and current research on database theory and techniques. Although the title might suggest otherwise, the constituency of this conference in recent years has extended well beyond the UK, indeed beyond European boundaries. This year the programme committee selected eleven papers and several posters for presentation at the conference from a large number of submissions. Of these eleven papers, three are from the UK, five from the European database research community, two from the USA and one from Australia. The programme committee also commissioned one additional UK contribution, in an area of growing industrial relevance.

BNCOD has been held at Cardiff once before, exactly ten years ago. Then, in 1988, at BNCOD-6 the main areas of research interest covered were object-oriented databases, temporal databases, and alternative data models; homogeneous distributed database systems were just beginning to be marketed and their heterogeneous relatives were still primarily a matter for research consideration. The scope and rate of change over the past decade, with the advent of the Internet and World Wide Web, the shift in emphasis from data to knowledge bases and the introduction of intelligent software agent technology into the development of distributed information systems, to name but three major advances, has been nothing short of staggering. Hence the main areas of research interest covered here are very different from those of 1988, as will become apparent.

Now, as then, the BNCOD paper presentations have been complemented by contributions from eminent invited speakers, both of whom on this occasion can boast two prior decades of major achievement in their respective fields of expertise. The first of these, Professor Malcolm Atkinson, to start the conference, reviews progress with providing orthogonal persistence for Java, in terms (mainly) of solid technical successes and residual political/commercial hurdles; his presentation includes a description and demonstration of a freely available operational prototype, PJama. The second, Jeffrey Walker of Tenfold Corporation, from his long experience at the forefront of the database and applications software industry, discusses outstanding productivity in his intriguingly titled presentation on "how to do a five-year applications development project in six months"!

Apart from these two invited keynote addresses and sessions devoted to introducing and viewing posters, the conference has six sessions spread over two days for the presentation of papers. In the first of these, on Middleware, Steiert and Zimmermann describe their JPMQ (Java Persistent Message Queues) system and Stephen Todd of IBM (at the request of the BNCOD programme committee) surveys progress in this important area of development from an industry

standpoint; the session concludes with a Panel Discussion on the merits of middleware.

The second session of papers addresses Life-Cycle aspects of database design. In this session, Griebel *et al.* consider the utilisation of repository technology for supporting design activities, then Preuner and Schrefl discuss the integration of views of object life-cycles in object-oriented database design, with particular relevance to their application in representing business processes.

The third papers session is on Association. It contains two papers: the first, by Omiecinski and Savasere, presents an efficient algorithm for mining association rules within the context of a dynamic database (i.e. where transactions can be added); and the second, by Kurniawati *et al.*, is concerned with improving the efficiency of the nearest-neighbour search operation using the weighted euclidean distance metric by calculating a bounding envelope of the nearest-neighbour query regions directly.

In the fourth session of papers, on Images, Nes and Kersten present their Acoi[1] query algebra for image retrieval systems, then Speegle *et al.* describe the meta-structure aspects of a method based on space-saving transformational representations to support multi-media editing in object-oriented image databases.

Heterogeneity is the subject of the fifth session of papers. Here, Karunaratna *et al.* discuss the establishment of a semantic knowledge base to assist integration of heterogeneous databases by helping users of a loosely-coupled federation of such databases to create views of and determine information available in the federation. This paper is followed by one from Conrad *et al.* which concentrates on an approach to deal with integrity constraints within the process of federated database design.

The sixth and final session of papers is on Languages, and again consists of two papers. The first, by Meredith and King, focuses on the identification of referentially transparent regions or scopes within an experimental variant of the functional database language FDL, called Relief, which is lazy and supports database updates. The second, by Paton and Sampaio, proposes a deductive object query language (DOQL) extension to the ODMG object database standard; DOQL is designed to conform to the main principles of ODMG compliant languages, thus providing a powerful complementary mechanism for use with ODMG databases.

Acknowledgements

As the list of editors indicates, the local organisation of BNCOD-16 has been very much a Cardiff team effort. As a team, we would like to thank the members of the programme committee for their work in reviewing, debating in committee and selecting the papers and posters for presentation at the conference. Special thanks are due to Professor Keith Jeffery for chairing the programme committee

[1] Amsterdam catalogue of images

meeting so effectively. The fact that every review of every paper was submitted in time for that meeting is thought to be something of a record and is to the credit of all the programme committee members. Thanks are also due to members of the BNCOD-15 Birkbeck College (London) organising committee for providing us with information and advice based on their previous year's experience, and for hosting this year's programme committee meeting.

We are indebted to other staff in the Department of Computer Science at Cardiff for their support in many aspects of administration of BNCOD-16: Anton Faulconbridge, Margaret Evans and Robert Evans provided invaluable help in this regard.

Sponsorship for the conference in the form of financial support and/or services offered free was supplied by several organisations and individuals both academic and industrial; this was most welcome in all cases and is gratefully acknowledged.

Cardiff University
April 1998

Nick Fiddian

Conference Committees

Programme Committee

K.G. Jeffery (Chair)	Rutherford Appleton Laboratory
S.M. Embury	Cardiff University
N.J. Fiddian	Cardiff University
C. Goble	University of Manchester
W.A. Gray	Cardiff University
J. Grimson	Trinity College, Dublin
J.G. Hughes	University of Ulster at Jordanstown
M. Jackson	University of Wolverhampton
M. Kay	ICL
G. Kemp	Aberdeen University
J. Kennedy	Napier University
J. Kerridge	Napier University
P.J.H. King	Birkbeck College
B. Lings	University of Exeter
R. Lucas	Keylink Computers
P. McBrien	King's College London
N.W. Paton	University of Manchester
A. Poulovassilis	King's College London
G. Sharman	IBM
M. Shave	Liverpool University
C. Small	SCO
S. Todd	IBM
M.H. Williams	Heriot-Watt University
M.F. Worboys	Keele University

Additional Referees

A. Dinn	Napier University
A.C. Jones	Cardiff University
P.R. Sampaio	University of Manchester
S. Tessaris	University of Manchester

Organising Committee

N.J. Fiddian (Chair)	Cardiff University
S.M. Embury	Cardiff University
M. Evans	Cardiff University
R. Evans	Cardiff University
A. Faulconbridge	Cardiff University
W.A. Gray	Cardiff University
A.C. Jones	Cardiff University

Steering Committee

W.A. Gray (Chair)	Cardiff University
N.J. Fiddian	Cardiff University
C. Goble	University of Manchester
P.M.D. Gray	University of Aberdeen
R. Johnson	Birkbeck College
J. Kennedy	Napier University
M.F. Worboys	Keele University

Table of Contents

Posters

JPMQ—An Advanced Persistent Message Queuing Service

Hans-Peter Steiert and Jürgen Zimmermann

Department of Computer Science, University of Kaiserslautern, P. O. Box 3049
D-67653 Kaiserslautern, Germany
{steiert, jnzimmer}@informatik.uni-kl.de
http://www.uni-kl.de/AG-Haerder/

Abstract. Message oriented middleware (MOM) services, based on the concept of persistent queues, are widely used for reliable asynchronous communication to deal with the disadvantages of the closely coupled communication model. This paper introduces some new functionality for queuing services (QS) in order to make them more powerful for the need of modern application programs and presents JPMQ, a sample implementation of these concepts. As such, JPMQ provides an object-oriented type system for messages, dynamic selection and ordering of messages as well as content-based priorities. Moreover, JPMQ offers transactional concepts in order to cope with concurrent access and failure recovery.

1 Introduction

Large-scale business applications are often comprised of autonomous, distributed programs, each of which performs a certain piece of work of a (more complex) task. In such domains, reliable communication is one of the primary necessities for correct task processing because the current state-of-progress has to be communicated between the programs. A closely coupled communication model based on a distributed two-phase commit protocol ([9]) allows programs to exchange data (i. e. the task's state-of-progress) in a reliable way. Usually a directed flow of information can be observed. Therefore, communication is sufficiently reliable/stable if the sending program can be 'sure' that the information sent will be received by the 'next-step' program.

This is a major issue addressed by so-called queuing services (QS), also known as message oriented middleware (MOM, [14]). These systems manage messages—as units of data to be exchanged—by using queues. After a client program has stored a message in a queue, it may continue processing without having to wait for the 'next-step' program to accept the message. Once a message is queued, the QS guarantees delivery, even when the target system is not reachable (either intentionally or due to failure) at the time of queuing ([2]). As for every base service, a seamless integration with application development and easy operation are necessary.

We want to address these topics by presenting a value-added QS, called JPMQ (*Java Persistent Message Queues*, [12], [18]). It provides semantically

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 1–18, 1998.

richer functionality, as for example queries on the contents of queues. Since this functionality is integrated into the enqueue and dequeue operations, application development can easily take advantage of these features.

This paper is organized in the following manner. In the next section, we give a short introduction into persistent queuing and the basic concepts realized in current queuing systems. Then we introduce some concepts we call "advanced queuing functionality", which go beyond the queuing facilities supported by current QS. Sect. 4 outlines our queuing system JPMQ, followed by the description of its implementation. We finish up with conclusions and a look into future work.

2 Persistent Queuing

Persistent queuing is a widely used technique for asynchronous communication between application programs. While messaging and synchronous communication can only be applied if the interacting programs run at the same time, the usage of a QS for communication relaxes this restriction.

The use of a QS simplifies the development of applications with asynchronous peer-to-peer communication, because it provides a reliable message buffer on the basis of queues. To offer an easy-to-use programming scheme, a QS integrates several concepts, namely those of messages, queues, and—most important in respect to quality of service concerns—an appropriate transaction semantics.

Queues can be considered from different perspectives, namely from the point of view of the QS and from that of the application programmer.

For the QS, queues are containers for arbitrary messages, which are under its control. Therefore, the QS takes care of all defined queues and their associated attributes. Queues accept an application program's storage and retrieval requests for messages which are commonly known as *enqueue* and *dequeue* operations.

From an application programmer's point of view, queues provide a helpful abstraction for distributed application development that uses programs in producer-consumer relationships. As such, producer and consumer do not have to know each other since the queue abstracts from a direct relationship. For a producer, a queue is a destination for its output data, while for a consumer it is a source of its input.

As we have already mentioned, the basic operations on queues are *enqueue* and *dequeue*. The enqueue operation is used by an application program to store a message into a specific queue. When an enqueue operation is processed successfully, the QS takes the message and guarantees its permanent storage. The counterpart of the enqueue operation is the dequeue operation. An application program uses the dequeue operation to request the delivery of a message from a specified queue. A successfully terminated dequeue operation releases the QS from controlling the delivered message. It may be possible to dequeue more than one message at once from the queue using a so-called "multiple dequeue" operation.

In order to enable application programs to differentiate messages according to their importance, a QS can allow explicit assignment of priority values along

with the enqueue operation. For messages with a priority value, a specialized option of the dequeue operation exists which removes messages from the queue in ascending or descending order of these priority values. Messages enqueued without any priority value are considered to be of a static priority.

Transactional concepts for operations upon queues are desirable for concurrent access and failure recovery ([9]). In the context of a QS, we will reconsider the ACID transaction paradigm ([10]) which has proven to be the most appropriate one.

Obviously the *durability* property must be provided by each QS. It ensures that enqueued messages do not vanish (until dequeue) and that dequeued messages are no longer accessible.

Atomicity is desirable in the sense that the entire set of dequeue and enqueue operations made during one transaction is performed successfully or none of them are. The consequence for the QS is, that effects of unsuccessfully terminated transactions (e. g. due to a program crash) will not show up.

Isolated execution is a means of coping with the requirement of atomicity. If all operations that an application program performs on queues are isolated from (i. e. not visible to) operations of other application programs, then those operations can be rolled back without affecting another application program's work.

With regard to *consistency*, the possibility of defining some consistency rules for messages is desirable. The simplest consistency requirement is surely that enqueued messages are not altered as long as the QS is responsible for them. Furthermore, some consistency rules could be defined for queued messages, e. g., to prevent the insertion of wrong data.

3 Advanced Queuing Functionality

The actual reason for the development of JPMQ was the need for so-called *advanced queuing functionality*, since rudimentary services such as enqueue or dequeue operations are already provided by today's products ([1], [4], [7]).

For the support of this advanced functionality, we will introduce complex message types which allow for the specification and control of the internal structure of messages to be exchanged. Starting from those complex messages, we propose several extensions based on the message contents: automatic priority assignment of enqueued messages, application-specific dequeue ordering and selective dequeuing. Another point we address are correctness issues guaranteed by the QS for queues in a multi-producer, multi-consumer environment. In the following, we want to discuss each of these extensions in more detail.

3.1 Complex Message Types

As already mentioned, a QS will be a useful middleware service for building distributed applications. Today's business applications are developed using object-oriented design methods ([3], [11]). The use of a semantically rich object model

in the design phase is often reflected by the usage of structured data in the implementation. Hence, an adequate representation of data transferred by messages is not a flat byte stream but a structured data type. This leads to the conclusion that the QS should be aware of the structure of the messages stored. In JPMQ, this is obtained by strict typing, which means that a message type is associated with every message. Thus, JPMQ has to provide a type system and a mechanism to define message types. The main design objective for the type system is to make it simple to use but flexible enough to define complex message types for almost every application domain.

JPMQ message types are defined using a three level hierarchy, which directly reflects the different abstraction levels. The topmost concept is that of a *message type* used for the strict typing of messages. A message type defines the unit of data exchange and has a service-wide unique identifier. As messages are complex structures, a message type consists of one or more *node types*. These node types group together a set of *attributes* that describe the different instances of this node type. A node type carries an identifier which must be unique within the message type. The attributes of a node type itself are again uniquely identified by their name, and can hold either *values* or *references*. Values are of simple type (e. g., strings, boolean or numerical values) whereas references represent relationships between different nodes. As a message type may contain more than one node type, one of them must be designated as the message type's *root node*. An instance of this node type becomes the only entry point (root) from which the remaining parts (nodes) of the message can be reached. A metamodel of our type system is shown in Fig. 1, an example message type definition in Fig. 2.

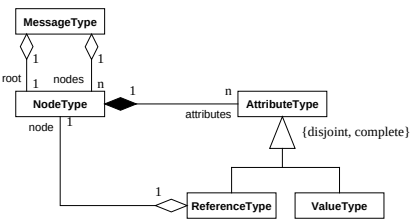


Fig. 1. Metamodel of the JPMQ type system (UML notation)

3.2 Automatic Priority Assignment

As JPMQ is aware of a message's internal structure, it can easily access the values stored within a message. A rather simple feature of JPMQ is automatic priority calculation and assignment for messages enqueued without an explicitly given priority value (see Sect. 2).

The priority value calculation is done by the means of a specific "priority calculation function" (PCF) which has to be specified in an OQL-like query language (see ODMG standard [5]). The PCF is evaluated while enqueuing the message and delivers a single value of type "real", which is taken as the message's

```
DEFINE MESSAGETYPE OrderMessage;
  NODE order;
    id: IDENT; date: DATE; lines: REFERENCE TO order_line;
  END;
  NODE order_line;
     line_no: INTEGER; amout: INTEGER; price: REAL; item: REFERENCE TO item;
  END;
  NODE item;
    id: IDENT; name: STRING; ean: INTEGER;
  END;
  ROOT order;
END;
```

Fig. 2. Example DDL statement for message type definition

priority. A PCF is always associated with a message type and a queue. The conjunction with a message type is obvious, since the query itself is tailored to the structure of that specific message type. The association with a queue allows for more freedom in defining special policies for different queues which accept the same message type.

3.3 Application-Specific Dequeue Ordering

Besides the rather static ordering of messages based on priority values assigned, JPMQ allows for application-specific dequeue ordering using a content-based sorting criterion. By this means, a reading application program can dynamically specify the order in which messages should be dequeued. For that purpose, we introduce a specialized dequeue operation that accepts a content-based "ordering function" (OF) which is conceptually comparable to a PCF except for the time of evaluation. While the PCF is statically defined once and evaluated along with the enqueue operation, the OF is dynamically issued and evaluated along with the dequeue operation to assign priority values to every message of the given message type residing in the queue used. In contrast to a PCF, an OF can deliver an ordered set of base types which is used to order the messages (see Sect. 5.3). Because PCFs make it possible to order all messages in a queue, for every message type we should get the same result type. As OFs are restricted to one message type, their usage can be simplified by allowing ordered sets as comparison values.

3.4 Selective Dequeue Operations

Besides dequeuing messages in a specific order (using PCFs or OFs), application programs often have to work exclusively with messages which satisfy certain predicates.

If only an ordinary dequeue operation is available, an application program has to dequeue all messages, select the qualifying ones and re-enqueue those that are not to be processed. Because all this has to be done within a single transaction (in order to avoid loss of messages), non-qualifying but read messages are not accessible to other application programs until commit, so concurrency

is reduced in most cases. Thus, a better solution would be to enable the QS by appropriate means to perform the selection 'nearby the queue', i. e., within the dequeue operation.

Providing this feature in JPMQ was no problem, because the message structure may serve as a foundation for the specification and evaluation of selection predicates. Thus, we introduce the "selective dequeue" operation. It accepts a "selection predicate" (SP), which (again) is specified in an OQL-like style. The SP is defined upon a concrete message type and evaluated during the dequeue operation on the messages of that type which are stored within the addressed queue. Only those messages which are included in the result set of the SP are delivered.

As a matter of fact, both the selective and sorting dequeue operations affect messages of a single type. While selection is evaluated on every single message, the ordering is done on a set of messages, so they can both be applied within a single dequeue operation which we call "selective ordering dequeue", as shown in the following example:

```
orders = outstandingOrdersQueue.dequeue
         ( "ORDER",
           no_of_orders,
           "count ( select ol.items from mt_OrderType om, order o, o.orderline ol )",
           "select om from mt_OrderType om, om.orders o where o.name = 'Smith'" );
```

3.5 Isolation Issues of JPMQ in a Multi-User Environment

So far, we didn't consider concurrency w. r. t. enqueuing and dequeuing messages. We now want to address the related problems by introducing different isolation levels. Programs enqueuing messages are called writers, those dequeuing message are called readers from now on.

JPMQ demands a queue to be in one of the four isolation levels "none", "ordered", "blockwise", or "ordered blockwise". These different levels stem from two independent factors, which determine the degree of the desired isolation. One factor is the application program from which messages originate and the other is the actual transaction (with visible transaction borders) used for enqueuing. The simplest (or lowest) isolation level ("none") means that there is no isolation between dequeuing transactions.

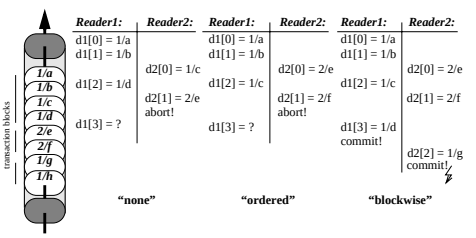


Fig. 3. Effects of different Isolation Levels

Let us, first of all, consider the case that there is just one reader. Without any ordering, he gets the messages in the order in which they are enqueued, so that we have a FIFO processing. It might happen that the reader gets messages enqueued by different writers in an intermixed way. Let us now imagine that there are two readers. Seen from the queue, this could lead to an intermixture of dequeue requests stemming from *Reader1* and *Reader2*, e. g., d1[0], d1[1], d2[0], d1[2], d2[1], d1[3] (of Fig. 3, left). The messages in this example are labelled with a number identifying the writer and a letter used as a message identifier.

If there is an abort of *Reader2* (as illustrated in Fig. 3, left) the QS might react by delivering "1/c" as result of the next dequeue operation (d1[3]) of *Reader1*, leading to a non-FIFO order of messages dequeued by *Reader1*. This is exactly what might happen when using isolation level "none". Hence, it is only useful, if there are no dependencies existing between different messages of the same writer.

To avoid such inconsistencies, we can isolate the two readers on the basis of the writers in such a way, that only one reader has access to the messages of one writer at a time, but may access messages of several writers. This level of isolation is called "ordered", which means that JPMQ guarantees a FIFO order for messages of one writer. Still, messages from different writers could be dequeued in an intermixed way, e. g. the result of d1[3] may be message "2/e" (Fig. 3, center), but *Reader2* is not able to receive any message from *Writer1* before *Reader1* has committed.

Up to isolation level "ordered" messages of different writers could be dequeued in in arbitrary intermixture. This may be sufficient in some cases, but for the realization of protocols with strong inter-message dependencies, neither "none" nor "ordered" is strong enough. Here, it makes more sense to take transaction boundaries into account: Only one reader gets exclusive access to messages of one (writing) transaction and has to dequeue all messages of this transaction. This means, the reader is only allowed to commit successfully if there are no more messages left from the writing transaction. The corresponding isolation level is called "blockwise" (Fig. 3, right).

Comparing the isolation levels "ordered" and "blockwise", we see that they address different problems. "Ordered" reserves all messages of a writer for the reader that accesses the first of his enqueued messages without taking (enqueuing) transactions into account. "Blockwise" only reserves the messages of a single enqueuing transaction, but imposes that the reader is aware of the transaction boundaries. So, "ordered" assures the writer's FIFO order, whereas "blockwise" reflects transactions. As such, it is a logical consequence that another isolation level exists, which combines those two concepts. This level is called "ordered blockwise" and has the following semantics: The first time a reader dequeues a message of a writer, it is guaranteed that this is the oldest message of that writer (that is currently in the queue) and, at the same time, exclusive access to all messages of this writer is granted. Similar to "blockwise", the reader can dequeue only messages of transactions of this writer in the order in which they are

enqueued. Moreover, a reader transaction must commit at one of the transaction boundaries which stem from the writers' transaction commit.

4 JPMQ System Architecture

In the following sections we will take a closer look at our implementation of JPMQ. The QS was designed as a middleware service used in a distributed computing infrastructure. We will explain the usage of JPMQ in such an environment and introduce its interfaces (Sect. 4.1). After that, we will examine the main components of JPMQ and how they cooperate to fulfil the requested services (Sect. 4.2). A more detailed description of implementation details is the main topic of Sect. 5.

4.1 Overall Architecture

In this section we will use the notion of "nodes" to differentiate between two QSs. The term *local node* is used as an alias for the JPMQ service to which an application program is currently connected. Accordingly, a *remote node* is a QS instance not known to the application program. Remote nodes may reside somewhere in the internet, in the local domain as well as on another continent.

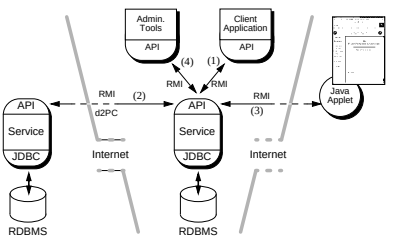


Fig. 4. External Architecure

JPMQ functionality may be claimed using one of four different interfaces (Fig. 4). The standard application programming interface (API, (1)) enables local application programs to access JPMQ. For communication between different instances of JPMQ, e. g. forwarding messages from a local to a remote node, a service-to-service interface (2) exists, which enables two JPMQ services to exchange messages. A distributed two-phase-commit protocol is used to ensure that no messages are lost. An applet-enabled interface (3) is provided in order to support lightweight clients, as for instance Web-clients or mobile computers. Electronic commerce will be a typical application using this kind of interface. In addition to the described interfaces, a fourth one is needed to create message types and queues in order to change default settings and to do maintenance (4).

4.2 Inside JPMQ

Let us now have a closer look inside JPMQ. Fig. 5 gives an overview of the main components, which cooperate together in order to achieve requested operations. We are exploiting the features of the Java-RMI mechanism for all client/server communication. Java-RMI hides the aspects of distributed object computing to the application developer ([17]). First of all, an application program has to

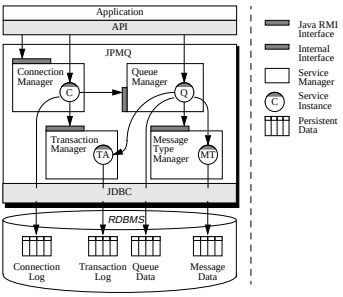


Fig. 5. Internal Architecure

connect to a JPMQ node. Then it may open some queues in order to enqueue or dequeue messages. Before issuing the first enqueue or dequeue operation a transaction has to be started.

The *connection manager* handles the incoming connect request to establish a connection to this node. This is achieved by creating a new connection object and passing it back to the application. This connection object privides the functionality to open or close queues. Requests to open a queue are forwarded to the *queue manager*, which is responsible for all queues managed by this JPMQ instance. The queue manager creates a new queue object, which is a transient representative of the persistent queue. As a result of the 'open queue' request, the corresponding queue object is passed to the applications.

Operations on queues are not allowed outside the scope of a running transaction. Transaction control is done by transaction objects. They are created by the *transaction manager* if a 'begin-of-transaction' operation is issued by the application. These requests are sent to the connection object and forwarded to the transaction manager. Whenever an enqueue or dequeue operation is performed by a queue object, the transaction object is notified and a log entry is written. In case of a system failure, these logs are taken in order to perform crash recovery. The implementation details of transaction control are covered in Sect. 5.2.

JPMQ is used to manage persistent queues with complex message types. A message consists of a header and a body. While the body contains application data, the header consists of information used by the queuing service itself. The header is always made of the same structure. Therefore, writing or reading the header information is done by the queue service. Because various message types have different internal structures, a specialized message type service object, which is managed by the *message type manager*, exists for every message type.

Reading and writing a message's application data is done by the corresponding message type service object.

5 Implementation

This section deals with implementation details of JPMQ. One very important design decision was to use an RDBMS as a storage layer. The advantages of using a RDBMS are discussed in Sect. 5.1. However, we do not consider the RDBMS transaction mechanism as sufficient for our QS. As already mentioned in Sect. 4.2, we decided to use our own transaction control. How it is based upon the ACID transactions provided by the RDBMS will be the topic of Sect. 5.2. Furthermore, we will discuss how JPMQ handles structured message types and how selective or ordering dequeue operations can utilize the query capabilities of RDBMSs (Sect. 5.3). In order to facilitate development, the object-oriented programming language Java is used for the implementation of JPMQ. Hence, we present our experiences using Java in Sect. 5.4.

5.1 The Storage Layer or Why to Use an RDBMS?

The advanced features we have already described did not cover one of the main facilities that any QS must handle: the reliable and persistent storage of messages. While messages can be persistently stored in various ways, we have chosen the proven technology of RDBMSs. This was done for several reasons, some of which stem from the capabilities of RDBMSs, and some of which allow us an easier and more powerful realization of the advanced features of JPMQ.

One of the reasons for a decision in favour of an RDBMS are the transaction properties, in our case, primarily duration and atomicity. If complete messages (with all nodes and their references) are stored in a database using a single database transaction, the RDBMS guarantees that, after a successful commit of that transaction, all the data is stored in a durable manner. Also, the atomicity property is useful, as a message is either stored completely or not, but not partially. Moreover, the RDBMS takes responsibility for data recovery actions to be taken after a system crash, so that JPMQ is grounded on a valid data basis. In Sect. 5.2, we describe how this is used as a bottom layer to build up our advanced transaction control.

An RDBMSs as the lowest level of data storage does not only bring us the advantage of ACID properties for single enqueue/dequeue operations, but also supports us in bridging the heterogeneity of the application domain in which JPMQ resides. Since the message contents are stored in attributes of database records (see Sect. 5.3), the RDBMS rather than the JPMQ is responsible for the physical representation of the values stored. Moreover, the use of SQL relieves us from becoming dependent on a specific RDBMS as long as it supports an appropriate implementation of the SQL standard ([15], [13]). This enables our system to run on top of almost every RDBMS which supports SQL if a suitable

JDBC driver is available. The JDBC interface is used to access the database as can be seen in Fig. 5.

One last, but not less important, reason for the usage of RDBMSs are the query facilities of relational database technology in the form of SQL. This allows us to utilize the RDBMS's query facilities to do value-based selection and ordering of messages for the advanced dequeue operations on certain message types.

5.2 Transaction Management

JPMQ allows grouping a set of enqueue and dequeue operations together as a transaction. Either all operations within this transaction are successfully done or none of them are. Additionally, a queue can be defined to ensure different levels w. r. t. the order of the messages inside the queue, even if several applications access the queue at the same time. As we have seen before, some of these properties are similar to database transactions.

Mapping every JPMQ transaction onto a database transaction would have been an easy way to implement JPMQ's transaction management. Unfortunately, this would lead to less concurrency, because every operation has to write data in tables holding administrative data, e. g. locking and logging information. Therefore, JPMQ maps every single queue operation onto a database transaction of their own, inheriting the durability properties of database transactions. This approach allows high concurrency in accessing the queue, but requires further steps to ensure atomicity and isolation. Before examining logging and recovery in more detail, we will discuss in the next section how JPMQ ensures isolation of concurrent readers and writers.

Isolation. JPMQ implements isolation by making messages invisible to transactions which are not allowed to access them. A flag, called *visibility state flag* (VSF), is used to hide messages. If a message is visible to all transactions, the VSF is set to '$\star$', otherwise it is set to the identifier of the transaction with exclusive access to the message.

The VSF is stored together with the *message identifier* (MID) and the *log state flag* (LSF) in the administrative data belonging to every message. This triple (MID, LF, VSF) is used to compose the message's state. The MID is built from the three parts S, T and M, whereby S is the identifier of the writer, e. g. the writing connection. T is the identifier of the transaction which enqueued this message. It is unique inside the connection S. The third part, M, is a unique identifier used to determine the message itself. The concatenation of S, T, and M—separated by '/'—forms the MID. We will refer to this representation of the MID when explaining the isolation protocols of the different isolation levels.

Fig. 6, left, shows the different message states which may occur while a message stays physically inside a queue with isolation level "none". After a writer 'a' has enqueued message 'n' in transaction 'a/1', the VSF is set to 'a/1' (message state $(\mathrm{a}/1/1, \mathrm{e}, \mathrm{a}/1)$). Now the message is only visible to operations performed

by transaction 'a/1'. If writer 'a' commits the transaction, the VSF is set to '$\star$' making the message visible to all other transactions (message state (a/1/1, c, $\star$)). In isolation level "none", no other operation will affect the VSF except for a rollback ($r_{a/1}$), which is described as part of the recovery. If isolation

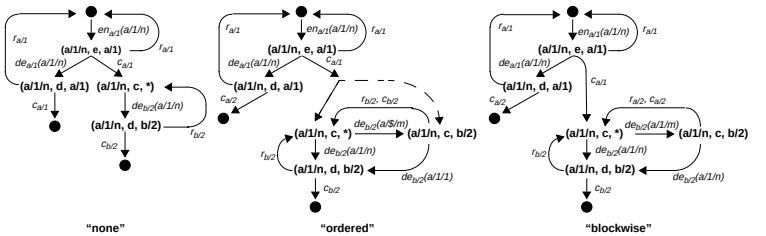


Fig. 6. Visibility Graphs

level "ordered" or "blockwise ordered" is used, the message state diagram is more complex, because the VSF may also be affected by additional operations. First, the commit of transaction 'a/1' may result either in state (a/1/n, c, $\star$) or (a/1/n, d, b/2). Second, a dequeue operation will affect the VSF, even if another message is read from the queue. Both result from the need to make all messages of a single writer invisible to other transactions, if transaction 'b/2' reads the first message of writer 'a' (Fig. 6, center).

Assume transaction '2' of reader 'b' reads message 'a/2/m'. In this case, the VSF of all messages 'a/$/$' is set to 'b/2', making these messages invisible to transactions other than 'b/2'. Here '$' may be any T or M in 'S/T/M'. If transaction 'a/1' commits after 'b/2' has read a message 'a/$/$' from the queue, the state of the message 'a/1/n' enqueued by 'a/1' is (a/1/n, c, b/2) instead of (a/1/n, c, $\star$).

The isolation protocol used in isolation level "blockwise" differs only a little bit from the one examined above. If a message is read, then all messages of the same transaction are invisible until the reading transaction commits (Fig. 6, right).

When implementing the message state diagrams presented in this section we exploit the set-oriented operations of the RDBMS used. Every arc of the diagram can be implemented using one query. The state of all messages is updated to the new state, by using the old state as the qualification criterion in the WHERE clause of the update (or delete) operation.

Notice that the order itself is not ensured by the isolation protocol. The messages are ordered on dequeue by using a sufficient order criterion along with the queries. In isolation level "none" or "ordered", all messages are sorted using their timestamp. If the isolation level is set to "blockwise", JPMQ orders the messages using the 'S/T' of the MID combined with a timestamp as an order criterion in the appropriate database queries. Hence, the levels "ordered" and "ordered blockwise" are reflected by the same message state diagram.

Logging and Recovery. As mentioned before, JPMQ has to track every queue operation. The corresponding log records are stored in different database tables. Because the log records are written in the same database transaction as the corresponding queue operation, it is ensured that the log is consistent with the application data, i. e., the messages written to the queue. After a crash occurs, the log is analysed. If there are operations of uncommitted transactions pending, undo recovery is necessary. There is no need for any redo recovery, because all operations of a queue transaction are reflected in the database after their commit. As messages cannot be changed while they are held under control of the QS and dequeued messages remain in the database until commit, there is no need for logging before-images. Instead, JPMQ writes log entries only for the operations performed upon the queue during a transaction into the log tables. Beside the VSF, the message state also carries a *log state flag* (LSF). This flag is used to mark the state of the message regarding to the log entries.

Take the following example. Assume the isolation level is set to "none" (Fig. 6, left). After a message is enqueued by transaction 'a/1', the LSF of this message is set to 'e'. Now, the system crashes before committing 'a/1'. So, all messages written by transaction 'a/1' must be removed from the queue. This means that all messages in message state (a/1/$, e, a/1) are deleted, an operation easily performed using the set-oriented operations of the RDBMS. Undo of dequeue operations is also easy. All messages dequeued by transaction 'b/2' are in message state (a/$/$, d, b/2). Hence, undo is done by updating the message state as shown in (Fig. 6, left) to the new state (a/$/$, c, $\star$). As we can see in the different message state diagrams some special actions have to be performed if isolation levels other than "none" are used or messages are enqueued and dequeued by the same transaction. For example, the VSF of messages not enqueued or dequeued by the transaction may have changed in order to ensure correct isolation. JPMQ has to undo these changes, too. In the state diagrams an 'r' marks all arcs which belong to undo operations. By examining the log entries, JPMQ decides which of these state transitions must be performed and how the set-oriented operations of the storage layer can be utilized.

5.3 Complex Message Types and Query Evaluation

In this section, we will describe how JPMQ exploits the data management facilities in order to store message types and to evaluate queries. For this reason, we need a mapping of the complex message types onto the relational data model, and a transformation of the OQL-like queries of JPMQ to SQL queries.

Let us first examine the mapping of the complex message types. Each message consists of a header and a body. The header holds the administrative information regarding the message. This is reflected by a special node added to the message type by JPMQ. It is always used as the root (header) of a message with a structure uniform for all message types and includes administrative data such as the priority value of the message, the lock and visibility flags, the message type name, and a timestamp of the enqueue time. JPMQ stores this data in a table representing the queue. As we have already seen, each message is given an

unique MID which is used as a primary key of this table. A foreign key is used to reference the root node of the application data which is stored in the body of a message type.

The database schema used to store application data may be built up from several tables. JPMQ creates a database table for each node type which is contained in the message type. The node type's attributes become columns within these tables and have an appropriate data type assigned. For the references between node types, extra tables are created because they may be (n:m)-relationships at the instance level. Hence, JPMQ creates five tables to store the message type defined in Fig. 2.

As already mentioned, we also need to map the JPMQ queries to the query language SQL. JPMQ uses queries to calculate the PCF and to provide a value-added dequeue operation with user defined ordering and selection criteria. The usage of an RDBMS enables JPMQ to delegate the evaluation of these queries to the storage layer. JPMQ provides a structural object-oriented type system with set-valued references. Hence, SQL as a JPMQ query language is insufficient, because application developers would need to know the mapping of message types to tables, thereby losing the advantages of a more object-oriented type model. Also, the SQL queries resulting from the mapping of message types to tables tend to be very complex. Therefore, we decided to use an OQL-like query language for JPMQ queries, which is aware of set-valued attributes. However, the JPMQ queries have to be mapped to SQL in order to exploit the query power provided by the storage layer.

Explaining the full JPMQ query language is beyond the scope of this paper. Nevertheless, we present some examples which should show that transformation of the queries requires some efforts but is not a critical task. For JPMQ queries we demand some restrictions in order to ease the process of query transformation. For example, the queries

```
select  *
from    mt_ordermessage mt, mt.root o, orderlines ol
where   ol in o.orderline
```

and

```
select  *
from    mt_ordermessage mt, mt.root o, o.orderlines ol
```

will lead to the same result. Yet, the first one is not a valid JPMQ query because of the free variable 'ol'. The 'from' clauses of a JPMQ query must span an directed acyclic graph, whereas the variables representing message nodes are the nodes and the references are the arcs. Free variables are variables which are not reachable in this graph starting at the root node and following the arcs.

Notice, that the node representing the administrative information of a message is always the anchor of a JPMQ query and is named 'mt_<message type name>'. The attribute 'root' references the root of the message type. Administrative information represented by attributes of node 'mt_<message type name>' can be used in queries too.

The example from Sect. 3.4 follows these restrictions. Executing the selective ordering dequeue operation means transforming OQL statements into SQL statements. First, the selection criterion is evaluated. Then message identifiers of the result set are sorted by the OF and read from the database. At last JPMQ reads the application data regarding to these identifiers. This leads to SQL statements similar to the one below:

```
select  q1.mid as mid, pcf.sort1 as sort1
from    q_orderqueue q1, (<OF>) as of
where   q1.mid = pcf.mid and q1.mid in (<SC>)
order by sort1
```

In order to evaluate the OF, the term `<OF>` is replaced by an SQL statement:

```
select  m2.mid as mid, count (i1.oid) as sort1
from    q_orderqueue q2, mt_ordermessage_order o1, mt_ordermessage_ref1 r1,
        mt_ordermessage_orderline ol1, mt_ordermessage_ref2 ref2, mt_ordermessage_item i1
where   q2.root = o1.oid and o1.oid = ref1.foid and ref1.soid = ol1.oid and
        ol1.oid = ref2.foid and ref2.soid = i1.oid
```

Accordingly, the term `<SC>` is replaced by the statement below:

```
select  q3.mid
from    q_orderqueue q3, order o2
where   q3.root = o2.oid and o2.name = 'Smith'
```

The selection and sorting of messages is a two-phase process. In the first phase, the qualifying messages and their order is determined, whereas in the second phase the actual data of the messages is read. In the second phase we take advantage of the MID stored in each record. As described before, JPMQ needs to access every table only once, reading all records which belong to the messages determined in the first phase. JPMQ creates main memory hash tables using the records representing references to ease the transformation of external storage references to Java references. Messages are held in a generic structure within the Java virtual machine.

5.4 Using Java

Our decision for Java([8]) as the implementation platform stems from several features of this programming language and its run-time system. The tight integration of the Java environment into the Internet is just one of the more important features.

With the concepts of RMI (*remote method invocation* [17]) along with *object serialization* as standards for Java, the gaps between JPMQ and applications are easily bridged in a portable manner. In the same line, the JDBC ([16]) standard provides a uniform, low level interface to any RDBMS which must—according to the specification of JDBC—support ANSI SQL-2 Entry Level ([15]). The usage of Java (and RDBMS) relieves us from dealing with heterogeneity aspects, since Java defines the physical layout of all types precisely and the concept of object serialization solves the problems of exchanging data between different platforms. Another feature supporting our decision for Java is the run-time extensibility of

applications. In this manner, we can extend the running JPMQ service with the functionality for new message types and appropriate message type services. In the remainder of this section, we introduce the generic data structures used to provide access to messages by application programs and the main functionality of the API.

Data structures. The application receives messages from the queue in the standard generic representation of JPMQ. Instances of the class `JPMQMessage` are used as entry points to the messages. Each node is represented by an instance of `JPMQMessageNode`, which is the superclass of `JPMQMessage`. Instances of `JPMQMessageNode` provide a generic late-binding interface for accessing the node's attributes. Relationships are built from instances of `JPMQReferenceCollection`. This class is a subclass of the Java class `Vector` and offers functionality for direct access to elements using an index as well as iterators to enumerate the records included. The generic classes come with the following operations:

```
JPMQValueType             JPMQMessageNode.getValue(String attrName);
JPMQReferenceCollection   JPMQMessageNode.getNodes(String attrName);
String                    JPMQMessageNode.getNodeName();
JPMQMessageNode           JPMQMessage.getRootNode();
String                    JPMQMessage.getTypeName();
Enumeration               JPMQReferenceCollection.getElements();
```

As application programmers should not be burdened with the internal mechanisms of the generic representation, an external representation for each message type may be generated using the message type definition. This external representation will act as a wrapper for the generic one in the sense that it presents a regular (Java) type to the application, but is a subclass of the generic type and provides functionality to switch between external and internal representation. In our example, five classes (`MTOrder` (subclass of `JPMQMessage`), `MOrder`, `MOrderline`, `MItem` (subclasses of `JPMQMessageNode`), `MOrderlineCollection`, `MItemCollection` (subclasses of `JPMQReferenceCollection`)) would have been generated. Let us examine `MTOrder` to give an idea of how the generated classes may be used. The interface will include methods to access the value-typed attributes as well as the set-valued attributes, as for example:

```
String                MOrder.getName();
void                  MOrder.setName(String name);
MOrderlineCollection  MOrder.getOrderline();
```

6 Conclusions and Outlook

In this paper we have introduced JPMQ which may be used as a base service to support reliable exchange of data between applications even through the Internet.

Middleware services must be easy to use and to support a programming model which fits into the application development environments. In the field of MOM, this means both supporting a type model matching the object-oriented

programming paradigm common in today's software projects and being applicable in a wide range of application domains. During the development of JPMQ these requirements have influenced our decision to support an application-independent type model for the exchange of structured data between applications.

JPMQ's structured type model was presented as the base of the advanced functionality provided by our MOM service. PCFs have been introduced, enabling the QS to schedule messages based on their contents. In addition, the dequeue operation was enhanced by introducing ordering and selection criteria. While selection criteria enable JPMQ-developed applications to decide which messages are needed at run-time, the ordering criteria makes it possible to change the order in which messages are received. Both are helpful extensions of the dequeue operation without loosing the simplicity of MOM services.

Our MOM service was designed to support reliable exchange of data. The ACID properties of database transactions were compared to the idea of queue transactions in order to detect similarities. Hence, JPMQ provides atomic queue transactions, even in the presence of failures. A logging scheme was introduced which enables JPMQ to recover a transaction-consistent state after a system crash. In addition, we argued the need of an isolation protocol to use JPMQ in multi-user environments with several application programs accessing one queue concurrently (as producers and/or consumers). Different isolation protocols were introduced in order to support a set of useful qualities of service.

We have also shown the advantages of using relational database technology. The properties of RDBMSs, i. e., ACID transactions and query interface, are useful with respect to the implementation of the service-specific operations and the transaction semantics of JPMQ. By using an RDBMS, message persistence is easy to implement. The ACID properties of database transactions enable us to achieve the service-specific isolation and logging protocols relying on an operation-consistent state regarding the data, the isolation state, and the log entries of JPMQ. Additionally, the set-oriented operations of an RDBMS are helpful to implement the isolation and logging protocols. Furthermore, we have taken advantage of the expressiveness of SQL to evaluate the PCFs as well as the ordering and selection criteria.

Java has been designed to ease development of distributed object systems (DOS) in the domain of Internet applications. JPMQ has been implemented using this technology, and we have shown that the provided concepts are really useful. Unfortunately, Java applications are still less efficient than, e. g., applications developed in C++. In order to allow the service to be transparently replaced by a more efficient implementation, the next step will be to enable access to JPMQ via CORBA interfaces. This will allow both the use of JPMQ by applications developed with other programming languages and the provision of a new service based on a different technology. We are very interested in the problems related to the use of a much less object-oriented technology, e. g., the C++ programming language.

Another aspect of future work is to achieve more failure tolerance in distributed object systems. Until now, a JPMQ node is identical to a JPMQ service. If the node fails, the service also fails. Hence, our goal is to build a JPMQ node by a group of JPMQ services. In case of a failure of one service, another service should be able to take over work.

Acknowledgments
We would like to thank Th. Härder and N. Ritter for their helpful comments on an earlier version of this paper.

References

1. B. Blakeley, H. Harris, R. Lewis: Messaging and Queuing Using MQI. McGraw-Hill, Inc. (1995) 3
2. P. A. Bernstein, M. Hsu, B. Mann: Implementing Recoverable Requests Using Queues. Proc. ACM SIGMOD (1990) 1
3. G. Booch: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings (1994) 3
4. P. A. Bernstein, E. Newcomer: Principles of Transaction Processing. Morgan Kaufmann Publishers, Inc. (1997) 3
5. R. G. G. Cattel, edt.: The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, Inc. (1996) 4
6. G. Coulouris, J. Dollimore, T. Kinderberg: Distributed Systems: Concepts and Design. Addison-Wesley Publishers Ltd. (1994)
7. Sybase, Inc.: dbQ User Guide & dbQ ReferenceGuide.
   Available at http://www.sybase.com/products/internet/dbq/ (June 1997) 3
8. J. Gosling, B. Joy, G. Steele: The Java Language Specification. Addison-Wesley, Inc. (1996) 15
9. J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, Inc. (1993) 1, 3
10. T. Härder, A. Reuter: Principles of Transaction Oriented Database Recovery. ACM Computing Surveys Vol. 15, No. 4 (1983) 3
11. I. Jacobson, M. Ericsson, A. Jacobson: The Object Advantage. ACP Press (1995) 3
12. M. Jotzo: JPQMS: Ein verteilter, objektorientierter Kommunikationsdienst für asynchrone Transaktionsverarbeitung. Diploma-Thesis (in German), Universität Kaiserslautern (December 1997) 1
13. J. Melton, A. R. Simon: Understanding the new SQL: a complete guide. Morgan Kaufmann Publishers, Inc. (1993) 10
14. R. Orfali, D. Harkey, J. Edward: The Essential Client/Server Survival Guide. John Wiley & Sons, Inc. (1996) 1
15. American National Standards Institute (ANSI): Database Language SQL. Document ANSI X3.135-192 10, 15
16. Sun Microsystems, Inc.: JDBC: A Java SQL API. Available at ftp://ftp.javasoft.com/pub/jdbc/jdbc-spec-0120.ps (March 1998) 15
17. Sun Microsystems, Inc.: RMI—Remote Methode Invocation. Available at http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/ (March 1998) 9, 15
18. H.-P. Steiert, J. Zimmermann: JPMQ—An Advanced Persistent Message Queuing Service (internal report).
    Available at http://www.uni-kl.de/AG-Haerder/publications/p1998.html 1

JPMQ—An Advanced Persistent Message Queuing Service

Hans-Peter Steiert and Jürgen Zimmermann

Department of Computer Science, University of Kaiserslautern, P. O. Box 3049
D-67653 Kaiserslautern, Germany
{steiert, jnzimmer}@informatik.uni-kl.de
http://www.uni-kl.de/AG-Haerder/

Abstract. Message oriented middleware (MOM) services, based on the concept of persistent queues, are widely used for reliable asynchronous communication to deal with the disadvantages of the closely coupled communication model. This paper introduces some new functionality for queuing services (QS) in order to make them more powerful for the need of modern application programs and presents JPMQ, a sample implementation of these concepts. As such, JPMQ provides an object-oriented type system for messages, dynamic selection and ordering of messages as well as content-based priorities. Moreover, JPMQ offers transactional concepts in order to cope with concurrent access and failure recovery.

1 Introduction

Large-scale business applications are often comprised of autonomous, distributed programs, each of which performs a certain piece of work of a (more complex) task. In such domains, reliable communication is one of the primary necessities for correct task processing because the current state-of-progress has to be communicated between the programs. A closely coupled communication model based on a distributed two-phase commit protocol ([9]) allows programs to exchange data (i. e. the task's state-of-progress) in a reliable way. Usually a directed flow of information can be observed. Therefore, communication is sufficiently reliable/stable if the sending program can be 'sure' that the information sent will be received by the 'next-step' program.

This is a major issue addressed by so-called queuing services (QS), also known as message oriented middleware (MOM, [14]). These systems manage messages—as units of data to be exchanged—by using queues. After a client program has stored a message in a queue, it may continue processing without having to wait for the 'next-step' program to accept the message. Once a message is queued, the QS guarantees delivery, even when the target system is not reachable (either intentionally or due to failure) at the time of queuing ([2]). As for every base service, a seamless integration with application development and easy operation are necessary.

We want to address these topics by presenting a value-added QS, called JPMQ (*Java Persistent Message Queues*, [12], [18]). It provides semantically

richer functionality, as for example queries on the contents of queues. Since this functionality is integrated into the enqueue and dequeue operations, application development can easily take advantage of these features.

This paper is organized in the following manner. In the next section, we give a short introduction into persistent queuing and the basic concepts realized in current queuing systems. Then we introduce some concepts we call "advanced queuing functionality", which go beyond the queuing facilities supported by current QS. Sect. 4 outlines our queuing system JPMQ, followed by the description of its implementation. We finish up with conclusions and a look into future work.

2 Persistent Queuing

Persistent queuing is a widely used technique for asynchronous communication between application programs. While messaging and synchronous communication can only be applied if the interacting programs run at the same time, the usage of a QS for communication relaxes this restriction.

The use of a QS simplifies the development of applications with asynchronous peer-to-peer communication, because it provides a reliable message buffer on the basis of queues. To offer an easy-to-use programming scheme, a QS integrates several concepts, namely those of messages, queues, and—most important in respect to quality of service concerns—an appropriate transaction semantics.

Queues can be considered from different perspectives, namely from the point of view of the QS and from that of the application programmer.

For the QS, queues are containers for arbitrary messages, which are under its control. Therefore, the QS takes care of all defined queues and their associated attributes. Queues accept an application program's storage and retrieval requests for messages which are commonly known as *enqueue* and *dequeue* operations.

From an application programmer's point of view, queues provide a helpful abstraction for distributed application development that uses programs in producer-consumer relationships. As such, producer and consumer do not have to know each other since the queue abstracts from a direct relationship. For a producer, a queue is a destination for its output data, while for a consumer it is a source of its input.

As we have already mentioned, the basic operations on queues are *enqueue* and *dequeue*. The enqueue operation is used by an application program to store a message into a specific queue. When an enqueue operation is processed successfully, the QS takes the message and guarantees its permanent storage. The counterpart of the enqueue operation is the dequeue operation. An application program uses the dequeue operation to request the delivery of a message from a specified queue. A successfully terminated dequeue operation releases the QS from controlling the delivered message. It may be possible to dequeue more than one message at once from the queue using a so-called "multiple dequeue" operation.

In order to enable application programs to differentiate messages according to their importance, a QS can allow explicit assignment of priority values along

with the enqueue operation. For messages with a priority value, a specialized option of the dequeue operation exists which removes messages from the queue in ascending or descending order of these priority values. Messages enqueued without any priority value are considered to be of a static priority.

Transactional concepts for operations upon queues are desirable for concurrent access and failure recovery ([9]). In the context of a QS, we will reconsider the ACID transaction paradigm ([10]) which has proven to be the most appropriate one.

Obviously the *durability* property must be provided by each QS. It ensures that enqueued messages do not vanish (until dequeue) and that dequeued messages are no longer accessible.

Atomicity is desirable in the sense that the entire set of dequeue and enqueue operations made during one transaction is performed successfully or none of them are. The consequence for the QS is, that effects of unsuccessfully terminated transactions (e. g. due to a program crash) will not show up.

Isolated execution is a means of coping with the requirement of atomicity. If all operations that an application program performs on queues are isolated from (i. e. not visible to) operations of other application programs, then those operations can be rolled back without affecting another application program's work.

With regard to *consistency*, the possibility of defining some consistency rules for messages is desirable. The simplest consistency requirement is surely that enqueued messages are not altered as long as the QS is responsible for them. Furthermore, some consistency rules could be defined for queued messages, e. g., to prevent the insertion of wrong data.

3 Advanced Queuing Functionality

The actual reason for the development of JPMQ was the need for so-called *advanced queuing functionality*, since rudimentary services such as enqueue or dequeue operations are already provided by today's products ([1], [4], [7]).

For the support of this advanced functionality, we will introduce complex message types which allow for the specification and control of the internal structure of messages to be exchanged. Starting from those complex messages, we propose several extensions based on the message contents: automatic priority assignment of enqueued messages, application-specific dequeue ordering and selective dequeuing. Another point we address are correctness issues guaranteed by the QS for queues in a multi-producer, multi-consumer environment. In the following, we want to discuss each of these extensions in more detail.

3.1 Complex Message Types

As already mentioned, a QS will be a useful middleware service for building distributed applications. Today's business applications are developed using object-oriented design methods ([3], [11]). The use of a semantically rich object model

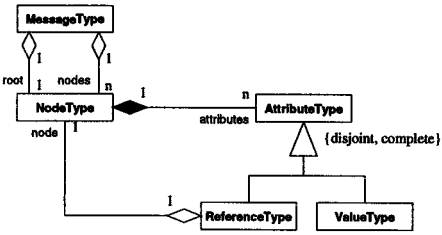


Fig. 1. Metamodel of the JPMQ type system (UML notation)

in the design phase is often reflected by the usage of structured data in the implementation. Hence, an adequate representation of data transferred by messages is not a flat byte stream but a structured data type. This leads to the conclusion that the QS should be aware of the structure of the messages stored. In JPMQ, this is obtained by strict typing, which means that a message type is associated with every message. Thus, JPMQ has to provide a type system and a mechanism to define message types. The main design objective for the type system is to make it simple to use but flexible enough to define complex message types for almost every application domain.

JPMQ message types are defined using a three level hierarchy, which directly reflects the different abstraction levels. The topmost concept is that of a *message type* used for the strict typing of messages. A message type defines the unit of data exchange and has a service-wide unique identifier. As messages are complex structures, a message type consists of one or more *node types*. These node types group together a set of *attributes* that describe the different instances of this node type. A node type carries an identifier which must be unique within the message type. The attributes of a node type itself are again uniquely identified by their name, and can hold either *values* or *references*. Values are of simple type (e. g., strings, boolean or numerical values) whereas references represent relationships between different nodes. As a message type may contain more than one node type, one of them must be designated as the message type's *root node*. An instance of this node type becomes the only entry point (root) from which the remaining parts (nodes) of the message can be reached. A metamodel of our type system is shown in Fig. 1, an example message type definition in Fig. 2.

3.2 Automatic Priority Assignment

As JPMQ is aware of a message's internal structure, it can easily access the values stored within a message. A rather simple feature of JPMQ is automatic priority calculation and assignment for messages enqueued without an explicitly given priority value (see Sect. 2).

The priority value calculation is done by the means of a specific "priority calculation function" (PCF) which has to be specified in an OQL-like query language (see ODMG standard [5]). The PCF is evaluated while enqueuing the message and delivers a single value of type "real", which is taken as the message's

```
DEFINE MESSAGETYPE OrderMessage;
  NODE order;
    id: IDENT; date: DATE; lines: REFERENCE TO order_line;
  END;
  NODE order_line;
     line_no: INTEGER; amout: INTEGER; price: REAL; item: REFERENCE TO item;
  END;
  NODE item;
    id: IDENT; name: STRING; ean: INTEGER;
  END;
  ROOT order;
END;
```

Fig. 2. Example DDL statement for message type definition

priority. A PCF is always associated with a message type and a queue. The conjunction with a message type is obvious, since the query itself is tailored to the structure of that specific message type. The association with a queue allows for more freedom in defining special policies for different queues which accept the same message type.

3.3 Application-Specific Dequeue Ordering

Besides the rather static ordering of messages based on priority values assigned, JPMQ allows for application-specific dequeue ordering using a content-based sorting criterion. By this means, a reading application program can dynamically specify the order in which messages should be dequeued. For that purpose, we introduce a specialized dequeue operation that accepts a content-based "ordering function" (OF) which is conceptually comparable to a PCF except for the time of evaluation. While the PCF is statically defined once and evaluated along with the enqueue operation, the OF is dynamically issued and evaluated along with the dequeue operation to assign priority values to every message of the given message type residing in the queue used. In contrast to a PCF, an OF can deliver an ordered set of base types which is used to order the messages (see Sect. 5.3). Because PCFs make it possible to order all messages in a queue, for every message type we should get the same result type. As OFs are restricted to one message type, their usage can be simplified by allowing ordered sets as comparison values.

3.4 Selective Dequeue Operations

Besides dequeuing messages in a specific order (using PCFs or OFs), application programs often have to work exclusively with messages which satisfy certain predicates.

If only an ordinary dequeue operation is available, an application program has to dequeue all messages, select the qualifying ones and re-enqueue those that are not to be processed. Because all this has to be done within a single transaction (in order to avoid loss of messages), non-qualifying but read messages

are not accessible to other application programs until commit, so concurrency is reduced in most cases. Thus, a better solution would be to enable the QS by appropriate means to perform the selection 'nearby the queue', i. e., within the dequeue operation.

Providing this feature in JPMQ was no problem, because the message structure may serve as a foundation for the specification and evaluation of selection predicates. Thus, we introduce the "selective dequeue" operation. It accepts a "selection predicate" (SP), which (again) is specified in an OQL-like style. The SP is defined upon a concrete message type and evaluated during the dequeue operation on the messages of that type which are stored within the addressed queue. Only those messages which are included in the result set of the SP are delivered.

As a matter of fact, both the selective and sorting dequeue operations affect messages of a single type. While selection is evaluated on every single message, the ordering is done on a set of messages, so they can both be applied within a single dequeue operation which we call "selective ordering dequeue", as shown in the following example:

```
orders = outstandingOrdersQueue.dequeue
         ( "ORDER",
           no_of_orders,
           "count ( select ol.items from mt_OrderType om, order o, o.orderline ol )",
           "select om from mt_OrderType om, om.orders o where o.name = 'Smith'" );
```

3.5 Isolation Issues of JPMQ in a Multi-user Environment

So far, we didn't consider concurrency w. r. t. enqueuing and dequeuing messages. We now want to address the related problems by introducing different isolation levels. Programs enqueuing messages are called writers, those dequeuing message are called readers from now on.

JPMQ demands a queue to be in one of the four isolation levels "none", "ordered", "blockwise", or "ordered blockwise". These different levels stem from two independent factors, which determine the degree of the desired isolation. One factor is the application program from which messages originate and the other is the actual transaction (with visible transaction borders) used for enqueuing. The simplest (or lowest) isolation level ("none") means that there is no isolation between dequeuing transactions.

Let us, first of all, consider the case that there is just one reader. Without any ordering, he gets the messages in the order in which they are enqueued, so that we have a FIFO processing. It might happen that the reader gets messages enqueued by different writers in an intermixed way. Let us now imagine that there are two readers. Seen from the queue, this could lead to an intermixture of dequeue requests stemming from *Reader1* and *Reader2*, e. g., d1[0], d1[1], d2[0], d1[2], d2[1], d1[3] (of Fig. 3, left). The messages in this example are labelled with a number identifying the writer and a letter used as a message identifier.

If there is an abort of *Reader2* (as illustrated in Fig. 3, left) the QS might react by delivering "1/c" as result of the next dequeue operation (d1[3]) of *Reader1*, leading to a non-FIFO order of messages dequeued by *Reader1*. This is exactly

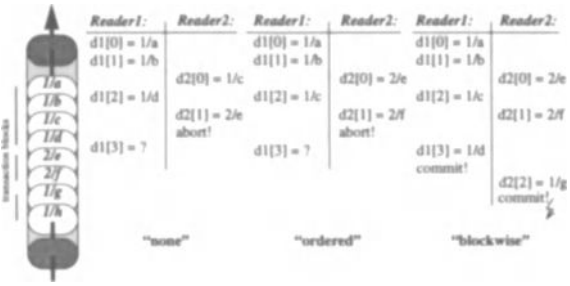


Fig. 3. Effects of different Isolation Levels

what might happen when using isolation level "none". Hence, it is only useful, if there are no dependencies existing between different messages of the same writer.

To avoid such inconsistencies, we can isolate the two readers on the basis of the writers in such a way, that only one reader has access to the messages of one writer at a time, but may access messages of several writers. This level of isolation is called "ordered", which means that JPMQ guarantees a FIFO order for messages of one writer. Still, messages from different writers could be dequeued in an intermixed way, e. g. the result of d1[3] may be message "2/e" (Fig. 3, center), but *Reader2* is not able to receive any message from *Writer1* before *Reader1* has committed.

Up to isolation level "ordered" messages of different writers could be dequeued in in arbitrary intermixture. This may be sufficient in some cases, but for the realization of protocols with strong inter-message dependencies, neither "none" nor "ordered" is strong enough. Here, it makes more sense to take transaction boundaries into account: Only one reader gets exclusive access to messages of one (writing) transaction and has to dequeue all messages of this transaction. This means, the reader is only allowed to commit successfully if there are no more messages left from the writing transaction. The corresponding isolation level is called "blockwise" (Fig. 3, right).

Comparing the isolation levels "ordered" and "blockwise", we see that they address different problems. "Ordered" reserves all messages of a writer for the reader that accesses the first of his enqueued messages without taking (enqueuing) transactions into account. "Blockwise" only reserves the messages of a single enqueuing transaction, but imposes that the reader is aware of the transaction boundaries. So, "ordered" assures the writer's FIFO order, whereas "blockwise" reflects transactions. As such, it is a logical consequence that another isolation level exists, which combines those two concepts. This level is called "ordered blockwise" and has the following semantics: The first time a reader dequeues a message of a writer, it is guaranteed that this is the oldest message of that writer (that is currently in the queue) and, at the same time, exclusive access to all messages of this writer is granted. Similar to "blockwise", the reader can dequeue only messages of transactions of this writer in the order in which they are

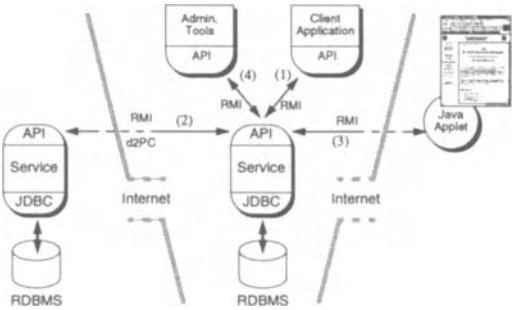


Fig. 4. External Architecure

enqueued. Moreover, a reader transaction must commit at one of the transaction boundaries which stem from the writers' transaction commit.

4 JPMQ System Architecture

In the following sections we will take a closer look at our implementation of JPMQ. The QS was designed as a middleware service used in a distributed computing infrastructure. We will explain the usage of JPMQ in such an environment and introduce its interfaces (Sect. 4.1). After that, we will examine the main components of JPMQ and how they cooperate to fulfil the requested services (Sect. 4.2). A more detailed description of implementation details is the main topic of Sect. 5.

4.1 Overall Architecture

In this section we will use the notion of "nodes" to differentiate between two QSs. The term *local node* is used as an alias for the JPMQ service to which an application program is currently connected. Accordingly, a *remote node* is a QS instance not known to the application program. Remote nodes may reside somewhere in the internet, in the local domain as well as on another continent.

JPMQ functionality may be claimed using one of four different interfaces (Fig. 4). The standard application programming interface (API, (1)) enables local application programs to access JPMQ. For communication between different instances of JPMQ, e. g. forwarding messages from a local to a remote node, a service-to-service interface (2) exists, which enables two JPMQ services to exchange messages. A distributed two-phase-commit protocol is used to ensure that no messages are lost. An applet-enabled interface (3) is provided in order to support lightweight clients, as for instance Web-clients or mobile computers. Electronic commerce will be a typical application using this kind of interface. In addition to the described interfaces, a fourth one is needed to create message types and queues in order to change default settings and to do maintenance (4).

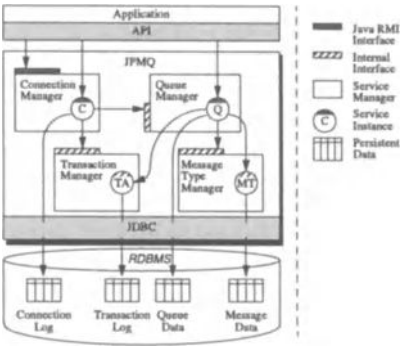


Fig. 5. Internal Architecure

4.2 Inside JPMQ

Let us now have a closer look inside JPMQ. Fig. 5 gives an overview of the main components, which cooperate together in order to achieve requested operations. We are exploiting the features of the Java-RMI mechanism for all client/server communication. Java-RMI hides the aspects of distributed object computing to the application developer ([17]).

First of all, an application program has to connect to a JPMQ node. Then it may open some queues in order to enqueue or dequeue messages. Before issuing the first enqueue or dequeue operation a transaction has to be started.

The *connection manager* handles the incoming connect request to establish a connection to this node. This is achieved by creating a new connection object and passing it back to the application. This connection object privides the functionality to open or close queues. Requests to open a queue are forwarded to the *queue manager*, which is responsible for all queues managed by this JPMQ instance. The queue manager creates a new queue object, which is a transient representative of the persistent queue. As a result of the 'open queue' request, the corresponding queue object is passed to the applications.

Operations on queues are not allowed outside the scope of a running transaction. Transaction control is done by transaction objects. They are created by the *transaction manager* if a 'begin-of-transaction' operation is issued by the application. These requests are sent to the connection object and forwarded to the transaction manager. Whenever an enqueue or dequeue operation is performed by a queue object, the transaction object is notified and a log entry is written. In case of a system failure, these logs are taken in order to perform crash recovery. The implementation details of transaction control are covered in Sect. 5.2.

JPMQ is used to manage persistent queues with complex message types. A message consists of a header and a body. While the body contains application data, the header consists of information used by the queuing service itself. The header is always made of the same structure. Therefore, writing or reading the header information is done by the queue service. Because various message types have different internal structures, a specialized message type service object,

which is managed by the *message type manager*, exists for every message type. Reading and writing a message's application data is done by the corresponding message type service object.

5 Implementation

This section deals with implementation details of JPMQ. One very important design decision was to use an RDBMS as a storage layer. The advantages of using a RDBMS are discussed in Sect. 5.1. However, we do not consider the RDBMS transaction mechanism as sufficient for our QS. As already mentioned in Sect. 4.2, we decided to use our own transaction control. How it is based upon the ACID transactions provided by the RDBMS will be the topic of Sect. 5.2. Furthermore, we will discuss how JPMQ handles structured message types and how selective or ordering dequeue operations can utilize the query capabilities of RDBMSs (Sect. 5.3). In order to facilitate development, the object-oriented programming language Java is used for the implementation of JPMQ. Hence, we present our experiences using Java in Sect. 5.4.

5.1 The Storage Layer or Why to Use an RDBMS?

The advanced features we have already described did not cover one of the main facilities that any QS must handle: the reliable and persistent storage of messages. While messages can be persistently stored in various ways, we have chosen the proven technology of RDBMSs. This was done for several reasons, some of which stem from the capabilities of RDBMSs, and some of which allow us an easier and more powerful realization of the advanced features of JPMQ.

One of the reasons for a decision in favour of an RDBMS are the transaction properties, in our case, primarily duration and atomicity. If complete messages (with all nodes and their references) are stored in a database using a single database transaction, the RDBMS guarantees that, after a successful commit of that transaction, all the data is stored in a durable manner. Also, the atomicity property is useful, as a message is either stored completely or not, but not partially. Moreover, the RDBMS takes responsibility for data recovery actions to be taken after a system crash, so that JPMQ is grounded on a valid data basis. In Sect. 5.2, we describe how this is used as a bottom layer to build up our advanced transaction control.

An RDBMSs as the lowest level of data storage does not only bring us the advantage of ACID properties for single enqueue/dequeue operations, but also supports us in bridging the heterogeneity of the application domain in which JPMQ resides. Since the message contents are stored in attributes of database records (see Sect. 5.3), the RDBMS rather than the JPMQ is responsible for the physical representation of the values stored. Moreover, the use of SQL relieves us from becoming dependent on a specific RDBMS as long as it supports an appropriate implementation of the SQL standard ([15], [13]). This enables our system to run on top of almost every RDBMS which supports SQL if a suitable

JDBC driver is available. The JDBC interface is used to access the database as can be seen in Fig. 5.

One last, but not less important, reason for the usage of RDBMSs are the query facilities of relational database technology in the form of SQL. This allows us to utilize the RDBMS's query facilities to do value-based selection and ordering of messages for the advanced dequeue operations on certain message types.

5.2 Transaction Management

JPMQ allows grouping a set of enqueue and dequeue operations together as a transaction. Either all operations within this transaction are successfully done or none of them are. Additionally, a queue can be defined to ensure different levels w.r.t. the order of the messages inside the queue, even if several applications access the queue at the same time. As we have seen before, some of these properties are similar to database transactions.

Mapping every JPMQ transaction onto a database transaction would have been an easy way to implement JPMQ's transaction management. Unfortunately, this would lead to less concurrency, because every operation has to write data in tables holding administrative data, e. g. locking and logging information. Therefore, JPMQ maps every single queue operation onto a database transaction of their own, inheriting the durability properties of database transactions. This approach allows high concurrency in accessing the queue, but requires further steps to ensure atomicity and isolation. Before examining logging and recovery in more detail, we will discuss in the next section how JPMQ ensures isolation of concurrent readers and writers.

Isolation. JPMQ implements isolation by making messages invisible to transactions which are not allowed to access them. A flag, called *visibility state flag* (VSF), is used to hide messages. If a message is visible to all transactions, the VSF is set to '$\star$', otherwise it is set to the identifier of the transaction with exclusive access to the message.

The VSF is stored together with the *message identifier* (MID) and the *log state flag* (LSF) in the administrative data belonging to every message. This triple (MID, LF, VSF) is used to compose the message's state. The MID is built from the three parts S, T and M, whereby S is the identifier of the writer, e. g. the writing connection. T is the identifier of the transaction which enqueued this message. It is unique inside the connection S. The third part, M, is a unique identifier used to determine the message itself. The concatenation of S, T, and M—separated by '/'—forms the MID. We will refer to this representation of the MID when explaining the isolation protocols of the different isolation levels.

Fig. 6, left, shows the different message states which may occur while a message stays physically inside a queue with isolation level "none". After a writer 'a' has enqueued message 'n' in transaction 'a/1', the VSF is set to 'a/1' (message state $(\mathrm{a/1/1, e, a/1})$). Now the message is only visible to operations performed

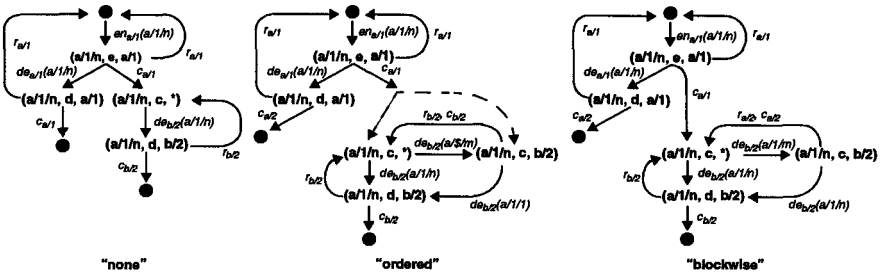


Fig. 6. Visibility Graphs

by transaction 'a/1'. If writer 'a' commits the transaction, the VSF is set to '$\star$' making the message visible to all other transactions (message state (a/1/1, c, $\star$)). In isolation level "none", no other operation will affect the VSF except for a rollback ($r_{a/1}$), which is described as part of the recovery.

If isolation level "ordered" or "blockwise ordered" is used, the message state diagram is more complex, because the VSF may also be affected by additional operations. First, the commit of transaction 'a/1' may result either in state (a/1/n, c, $\star$) or (a/1/n, d, b/2). Second, a dequeue operation will affect the VSF, even if another message is read from the queue. Both result from the need to make all messages of a single writer invisible to other transactions, if transaction 'b/2' reads the first message of writer 'a' (Fig. 6, center).

Assume transaction '2' of reader 'b' reads message 'a/2/m'. In this case, the VSF of all messages 'a/$/$' is set to 'b/2', making these messages invisible to transactions other than 'b/2'. Here '$' may be any T or M in 'S/T/M'. If transaction 'a/1' commits after 'b/2' has read a message 'a/$/$' from the queue, the state of the message 'a/1/n' enqueued by 'a/1' is (a/1/n, c, b/2) instead of (a/1/n, c, $\star$).

The isolation protocol used in isolation level "blockwise" differs only a little bit from the one examined above. If a message is read, then all messages of the same transaction are invisible until the reading transaction commits (Fig. 6, right).

When implementing the message state diagrams presented in this section we exploit the set-oriented operations of the RDBMS used. Every arc of the diagram can be implemented using one query. The state of all messages is updated to the new state, by using the old state as the qualification criterion in the WHERE clause of the update (or delete) operation.

Notice that the order itself is not ensured by the isolation protocol. The messages are ordered on dequeue by using a sufficient order criterion along with the queries. In isolation level "none" or "ordered", all messages are sorted using their timestamp. If the isolation level is set to "blockwise", JPMQ orders the messages using the 'S/T' of the MID combined with a timestamp as an order criterion in the appropriate database queries. Hence, the levels "ordered" and "ordered blockwise" are reflected by the same message state diagram.

Logging and Recovery. As mentioned before, JPMQ has to track every queue operation. The corresponding log records are stored in different database tables. Because the log records are written in the same database transaction as the corresponding queue operation, it is ensured that the log is consistent with the application data, i. e., the messages written to the queue. After a crash occurs, the log is analysed. If there are operations of uncommitted transactions pending, undo recovery is necessary. There is no need for any redo recovery, because all operations of a queue transaction are reflected in the database after their commit. As messages cannot be changed while they are held under control of the QS and dequeued messages remain in the database until commit, there is no need for logging before-images. Instead, JPMQ writes log entries only for the operations performed upon the queue during a transaction into the log tables. Beside the VSF, the message state also carries a *log state flag* (LSF). This flag is used to mark the state of the message regarding to the log entries.

Take the following example. Assume the isolation level is set to "none" (Fig. 6, left). After a message is enqueued by transaction 'a/1', the LSF of this message is set to 'e'. Now, the system crashes before committing 'a/1'. So, all messages written by transaction 'a/1' must be removed from the queue. This means that all messages in message state (a/1/$, e, a/1) are deleted, an operation easily performed using the set-oriented operations of the RDBMS. Undo of dequeue operations is also easy. All messages dequeued by transaction 'b/2' are in message state (a/$/$, d, b/2). Hence, undo is done by updating the message state as shown in (Fig. 6, left) to the new state (a/$/$, c, $\star$). As we can see in the different message state diagrams some special actions have to be performed if isolation levels other than "none" are used or messages are enqueued and dequeued by the same transaction. For example, the VSF of messages not enqueued or dequeued by the transaction may have changed in order to ensure correct isolation. JPMQ has to undo these changes, too. In the state diagrams an 'r' marks all arcs which belong to undo operations. By examining the log entries, JPMQ decides which of these state transitions must be performed and how the set-oriented operations of the storage layer can be utilized.

5.3 Complex Message Types and Query Evaluation

In this section, we will describe how JPMQ exploits the data management facilities in order to store message types and to evaluate queries. For this reason, we need a mapping of the complex message types onto the relational data model, and a transformation of the OQL-like queries of JPMQ to SQL queries.

Let us first examine the mapping of the complex message types. Each message consists of a header and a body. The header holds the administrative information regarding the message. This is reflected by a special node added to the message type by JPMQ. It is always used as the root (header) of a message with a structure uniform for all message types and includes administrative data such as the priority value of the message, the lock and visibility flags, the message type name, and a timestamp of the enqueue time. JPMQ stores this data in a table representing the queue. As we have already seen, each message is given an

unique MID which is used as a primary key of this table. A foreign key is used to reference the root node of the application data which is stored in the body of a message type.

The database schema used to store application data may be built up from several tables. JPMQ creates a database table for each node type which is contained in the message type. The node type's attributes become columns within these tables and have an appropriate data type assigned. For the references between node types, extra tables are created because they may be (n:m)-relationships at the instance level. Hence, JPMQ creates five tables to store the message type defined in Fig. 2.

As already mentioned, we also need to map the JPMQ queries to the query language SQL. JPMQ uses queries to calculate the PCF and to provide a value-added dequeue operation with user defined ordering and selection criteria. The usage of an RDBMS enables JPMQ to delegate the evaluation of these queries to the storage layer. JPMQ provides a structural object-oriented type system with set-valued references. Hence, SQL as a JPMQ query language is insufficient, because application developers would need to know the mapping of message types to tables, thereby losing the advantages of a more object-oriented type model. Also, the SQL queries resulting from the mapping of message types to tables tend to be very complex. Therefore, we decided to use an OQL-like query language for JPMQ queries, which is aware of set-valued attributes. However, the JPMQ queries have to be mapped to SQL in order to exploit the query power provided by the storage layer.

Explaining the full JPMQ query language is beyond the scope of this paper. Nevertheless, we present some examples which should show that transformation of the queries requires some efforts but is not a critical task. For JPMQ queries we demand some restrictions in order to ease the process of query transformation. For example, the queries

```
select  *
from    mt_ordermessage mt, mt.root o, orderlines ol
where   ol in o.orderline
```

and

```
select  *
from    mt_ordermessage mt, mt.root o, o.orderlines ol
```

will lead to the same result. Yet, the first one is not a valid JPMQ query because of the free variable 'ol'. The 'from' clauses of a JPMQ query must span an directed acyclic graph, whereas the variables representing message nodes are the nodes and the references are the arcs. Free variables are variables which are not reachable in this graph starting at the root node and following the arcs.

Notice, that the node representing the administrative information of a message is always the anchor of a JPMQ query and is named 'mt_<message type name>'. The attribute 'root' references the root of the message type. Administrative information represented by attributes of node 'mt_<message type name>' can be used in queries too.

The example from Sect. 3.4 follows these restrictions. Executing the selective ordering dequeue operation means transforming OQL statements into SQL statements. First, the selection criterion is evaluated. Then message identifiers of the result set are sorted by the OF and read from the database. At last JPMQ reads the application data regarding to these identifiers. This leads to SQL statements similar to the one below:

```
select  q1.mid as mid, pcf.sort1 as sort1
from    q_orderqueue q1, (<OF>) as of
where   q1.mid = pcf.mid and q1.mid in (<SC>)
order by sort1
```

In order to evaluate the OF, the term <OF> is replaced by an SQL statement:

```
select  m2.mid as mid, count (i1.oid) as sort1
from    q_orderqueue q2, mt_ordermessage_order o1, mt_ordermessage_ref1 r1,
        mt_ordermessage_orderline ol1, mt_ordermessage_ref2 ref2, mt_ordermessage_item i1
where   q2.root = o1.oid and o1.oid = ref1.foid and ref1.soid = ol1.oid and
        ol1.oid = ref2.foid and ref2.soid = i1.oid
```

Accordingly, the term <SC> is replaced by the statement below:

```
select  q3.mid
from    q_orderqueue q3, order o2
where   q3.root = o2.oid and o2.name = 'Smith'
```

The selection and sorting of messages is a two-phase process. In the first phase, the qualifying messages and their order is determined, whereas in the second phase the actual data of the messages is read. In the second phase we take advantage of the MID stored in each record. As described before, JPMQ needs to access every table only once, reading all records which belong to the messages determined in the first phase. JPMQ creates main memory hash tables using the records representing references to ease the transformation of external storage references to Java references. Messages are held in a generic structure within the Java virtual machine.

5.4 Using Java

Our decision for Java([8]) as the implementation platform stems from several features of this programming language and its run-time system. The tight integration of the Java environment into the Internet is just one of the more important features.

With the concepts of RMI (*remote method invocation* [17]) along with *object serialization* as standards for Java, the gaps between JPMQ and applications are easily bridged in a portable manner. In the same line, the JDBC ([16]) standard provides a uniform, low level interface to any RDBMS which must—according to the specification of JDBC—support ANSI SQL-2 Entry Level ([15]). The usage of Java (and RDBMS) relieves us from dealing with heterogeneity aspects, since Java defines the physical layout of all types precisely and the concept of object serialization solves the problems of exchanging data between different platforms. Another feature supporting our decision for Java is the run-time extensibility of

applications. In this manner, we can extend the running JPMQ service with the functionality for new message types and appropriate message type services. In the remainder of this section, we introduce the generic data structures used to provide access to messages by application programs and the main functionality of the API.

Data structures. The application receives messages from the queue in the standard generic representation of JPMQ. Instances of the class `JPMQMessage` are used as entry points to the messages. Each node is represented by an instance of `JPMQMessageNode`, which is the superclass of `JPMQMessage`. Instances of `JPMQMessageNode` provide a generic late-binding interface for accessing the node's attributes. Relationships are built from instances of `JPMQReferenceCollection`. This class is a subclass of the Java class `Vector` and offers functionality for direct access to elements using an index as well as iterators to enumerate the records included. The generic classes come with the following operations:

```
JPMQValueType            JPMQMessageNode.getValue(String attrName);
JPMQReferenceCollection  JPMQMessageNode.getNodes(String attrName);
String                   JPMQMessageNode.getNodeName();
JPMQMessageNode          JPMQMessage.getRootNode();
String                   JPMQMessage.getTypeName();
Enumeration              JPMQReferenceCollection.getElements();
```

As application programmers should not be burdened with the internal mechanisms of the generic representation, an external representation for each message type may be generated using the message type definition. This external representation will act as a wrapper for the generic one in the sense that it presents a regular (Java) type to the application, but is a subclass of the generic type and provides functionality to switch between external and internal representation. In our example, five classes (`MTOrder` (subclass of `JPMQMessage`), `MOrder`, `MOrderline`, `MItem` (subclasses of `JPMQMessageNode`), `MOrderlineCollection`, `MItemCollection` (subclasses of `JPMQReferenceCollection`)) would have been generated. Let us examine `MTOrder` to give an idea of how the generated classes may be used. The interface will include methods to access the value-typed attributes as well as the set-valued attributes, as for example:

```
String                MOrder.getName();
void                  MOrder.setName(String name);
MOrderlineCollection  MOrder.getOrderline();
```

6 Conclusions and Outlook

In this paper we have introduced JPMQ which may be used as a base service to support reliable exchange of data between applications even through the Internet.

Middleware services must be easy to use and to support a programming model which fits into the application development environments. In the field of MOM, this means both supporting a type model matching the object-oriented

programming paradigm common in today's software projects and being applicable in a wide range of application domains. During the development of JPMQ these requirements have influenced our decision to support an application-independent type model for the exchange of structured data between applications.

JPMQ's structured type model was presented as the base of the advanced functionality provided by our MOM service. PCFs have been introduced, enabling the QS to schedule messages based on their contents. In addition, the dequeue operation was enhanced by introducing ordering and selection criteria. While selection criteria enable JPMQ-developed applications to decide which messages are needed at run-time, the ordering criteria makes it possible to change the order in which messages are received. Both are helpful extensions of the dequeue operation without loosing the simplicity of MOM services.

Our MOM service was designed to support reliable exchange of data. The ACID properties of database transactions were compared to the idea of queue transactions in order to detect similarities. Hence, JPMQ provides atomic queue transactions, even in the presence of failures. A logging scheme was introduced which enables JPMQ to recover a transaction-consistent state after a system crash. In addition, we argued the need of an isolation protocol to use JPMQ in multi-user environments with several application programs accessing one queue concurrently (as producers and/or consumers). Different isolation protocols were introduced in order to support a set of useful qualities of service.

We have also shown the advantages of using relational database technology. The properties of RDBMSs, i. e., ACID transactions and query interface, are useful with respect to the implementation of the service-specific operations and the transaction semantics of JPMQ. By using an RDBMS, message persistence is easy to implement. The ACID properties of database transactions enable us to achieve the service-specific isolation and logging protocols relying on an operation-consistent state regarding the data, the isolation state, and the log entries of JPMQ. Additionally, the set-oriented operations of an RDBMS are helpful to implement the isolation and logging protocols. Furthermore, we have taken advantage of the expressiveness of SQL to evaluate the PCFs as well as the ordering and selection criteria.

Java has been designed to ease development of distributed object systems (DOS) in the domain of Internet applications. JPMQ has been implemented using this technology, and we have shown that the provided concepts are really useful. Unfortunately, Java applications are still less efficient than, e. g., applications developed in C++. In order to allow the service to be transparently replaced by a more efficient implementation, the next step will be to enable access to JPMQ via CORBA interfaces. This will allow both the use of JPMQ by applications developed with other programming languages and the provision of a new service based on a different technology. We are very interested in the problems related to the use of a much less object-oriented technology, e. g., the C++ programming language.

Another aspect of future work is to achieve more failure tolerance in distributed object systems. Until now, a JPMQ node is identical to a JPMQ service. If the node fails, the service also fails. Hence, our goal is to build a JPMQ node by a group of JPMQ services. In case of a failure of one service, another service should be able to take over work.

Acknowledgments

We would like to thank Th. Härder and N. Ritter for their helpful comments on an earlier version of this paper.

References

1. B. Blakeley, H. Harris, R. Lewis: Messaging and Queuing Using MQI. McGraw-Hill, Inc. (1995)
2. P. A. Bernstein, M. Hsu, B. Mann: Implementing Recoverable Requests Using Queues. Proc. ACM SIGMOD (1990)
3. G. Booch: Object-Oriented Analysis and Design with Applications. Benjamin/Cummings (1994)
4. P. A. Bernstein, E. Newcomer: Principles of Transaction Processing. Morgan Kaufmann Publishers, Inc. (1997)
5. R. G. G. Cattel, edt.: The Object Database Standard: ODMG-93. Morgan Kaufmann Publishers, Inc. (1996)
6. G. Coulouris, J. Dollimore, T. Kinderberg: Distributed Systems: Concepts and Design. Addison-Wesley Publishers Ltd. (1994)
7. Sybase, Inc.: dbQ User Guide & dbQ ReferenceGuide.
   Available at http://www.sybase.com/products/internet/dbq/ (June 1997)
8. J. Gosling, B. Joy, G. Steele: The Java Language Specification. Addison-Wesley, Inc. (1996)
9. J. Gray, A. Reuter: Transaction Processing: Concepts and Techniques. Morgan Kaufmann Publishers, Inc. (1993)
10. T. Härder, A. Reuter: Principles of Transaction Oriented Database Recovery. ACM Computing Surveys Vol. 15, No. 4 (1983)
11. I. Jacobson, M. Ericsson, A. Jacobson: The Object Advantage. ACP Press (1995)
12. M. Jotzo: JPQMS: Ein verteilter, objektorientierter Kommunikationsdienst für asynchrone Transaktionsverarbeitung. Diploma-Thesis (in German), Universität Kaiserslautern (December 1997)
13. J. Melton, A. R. Simon: Understanding the new SQL: a complete guide. Morgan Kaufmann Publishers, Inc. (1993)
14. R. Orfali, D. Harkey, J. Edward: The Essential Client/Server Survival Guide. John Wiley & Sons, Inc. (1996)
15. American National Standards Institute (ANSI): Database Language SQL. Document ANSI X3.135-192
16. Sun Microsystems, Inc.: JDBC: A Java SQL API. Available at ftp://ftp.javasoft.com/pub/jdbc/jdbc-spec-0120.ps (March 1998)
17. Sun Microsystems, Inc.: RMI—Remote Methode Invocation. Available at http://www.javasoft.com/products/jdk/1.1/docs/guide/rmi/ (March 1998)
18. H.-P. Steiert, J. Zimmermann: JPMQ—An Advanced Persistent Message Queuing Service (internal report).
    Available at http://www.uni-kl.de/AG-Haerder/publications/p1998.html

A Repository to Support Transparency in Database Design

Gerrit Griebel[1]**, Brian Lings[2], and Björn Lundell[1]

[1] University of Skövde, Sweden, Department of Computer Science
(bjorn.lundell@ida.his.se)
[2] University of Exeter, UK, Department of Computer Science
(brian@dcs.exeter.ac.uk)

Abstract. In this paper we address aspects of traceability in Information Systems design, particularly with those methodologies utilizing rich, and possibly multiple (behavioural) models. We consider this issue within the broader context of IS development, and focus on the potential for communication among human actors. Specifically, we consider the question of repository design for supporting design activities. We illustrate this through a Web-based visualization tool for Extended Entity-Relationship (EER) modeling represented in a repository based on the Case Data Interchange Format (CDIF). Such a tool offers a degree of independence with respect to CASE tools used within an organization, and validation methods chosen by an organization.

1 Introduction

1.1 Model Transparency

Traceability in Information Systems design, particularly with those methodologies utilizing rich, and possibly multiple (behavioural) models, is a complex issue. The aspect of traceability of concern in this paper, sometimes referred to as modeling transparency, is considered as an absolute requirement for advanced systems development [3].

Within the broader context of IS development it requires correlation of design decisions across different sub-models, and between data sets and meta data, on an ongoing basis. A number of different tools may be utilized, including CASE tools offering ER to SQL mapping and external validation tools. One way forward is the utilization of repository technology for supporting all design and maintenance activities. Such a repository would be designed to store all available modeling information, and all information necessary to allow transparency in the refinement process and with respect to dependencies between models. We have chosen to explore this idea through an investigation of the use of the Case Data Interchange Format (CDIF), initially for defining Extended Entity Relationship (EER) models, and latterly with respect to multiple models in the context of the

** current address: Soester Straße 48, 20099 Hamburg, Germany (gg@tron.ppp.de)

F3 methodology [6]. All the work so far has concentrated on the design phase of the life cycle; the support of maintenance activities forms a part of ongoing work. For this study, CDIF was chosen for its extensibility. We note, however, that the investigators within the F3 project had suggested the possible use of CDIF for our purposes: "We strongly suggest the CDIF standard is considered for F3 and its inter-tool communication." [5, p. 39]

We observe two main approaches to enhancing ER with respect to behavioural aspects at the conceptual level. Firstly, some contributions have added constructs to give a particular variant of EER, e.g. rules in [19] (anticipating active databases). A consequence of this approach is an increased complexity within diagrams, something which potentially is a hindering factor for its use in practice. Secondly, others suggest a set of different models to be used in conjunction at the conceptual level (as in F3). A consequence of this approach is that the number of inter-relationships that need to be maintained between related concepts in different models increases, potentially exponentially, with the number of models used.

In addition to 'extended model' and 'multi-model' approaches, we envisage a need for enhanced traceability between the conceptual level models and those models used at earlier stages of the IS development process. Our assumption here is that as the complexity of models increases, so will the variation in expectations and needs between users [13]. To address this difficulty, we anticipate that a meta-model approach (repository) will allow us to handle this complexity, something which we are currently exploring in the context of two (of the total of five) different F3 models with preserved traceability between the Enterprise Model level and the conceptual model.

1.2 Populated Models

We believe that, as a complement to communication at the level of models, techniques for browsing, visualization, animation etc. (see e.g. [4] and [12]) using populated models might significantly improve end-users' understanding of behavioural models.

Sample data or prerecorded data is often available before a development starts; investigation of the domain is facilitated by this data, which is often not used until installation of the developed database software.

When data is incorporated, a modeling tool can provide visual feedback of the dynamic implications of a schema and its integrity constraints (IC). This is useful for understanding an application database, since most DBMS allow predefined behaviour to maintain the integrity of data. This is particularly of value in the context of EER-Modeling, because relational IC can be automatically derived from EER model properties to lose as little model semantics as possible during mapping.

1.3 Aim of the Research

The concern of this work is therefore threefold:

1. To investigate mapping procedures with respect to preservation of EER model semantics
2. to design a repository suitable for storing all model representations and their interrelationships and
3. to propose a visualization architecture based on the repository and target application database to allow for visualization of the meta-data and the data at different levels.

The result is a proposal for a repository based visualization architecture which explicitly relates the application database data to the abstract relational level and to the EER level. Different aspects of the design and the refinements chosen can be traced in a comprehensible fashion. The focus of this work is design. The modeling tool S-Designor [15] was used to build the CDIF repository structure and to generate sample models. Java programs were written to exemplify access and visualization of the repository content [11].

2 The Layered Approach

We consider a three layered approach to designing schemas for DBMS as represented by the three boxes in the middle column of Figure 1.

A central point of this work is to allow not only a transparency of the mapping from EER to relational, but also its inverse. Commonly only the forward mapping from the EER to the relational model is considered in the design process. This is insufficient to support general visual browsing with genuine traceability. Appropriate structures must be found for storing this bidirectional interrelationship (Meta-Schema, left column of Figure 1). Structural mapping procedures are well covered by the literature (as in [9]). With the use of modern tools, most of the mapping can be derived automatically, while the designer can give explicit guidance, for example, on how to migrate candidate keys to referencing relations or how to map supertype/subtype or 1:1 relationships.

We focus on the production of a relational model which reflects the inherent constraints in an EER model with little loss of semantics. Some EER IC properties (e.g. disjoint relationships) cannot be expressed in the abstract relational layer, but can be modeled by trigger sets of a target database system. Therefore access paths from the EER information to the SQL-layer are necessary. Update propagation properties (cascade, restrict, update, set null and set default, comparable with those in [14]) which define the behaviour of the database in case of violation of IC are not included in common EER notations. These relational properties can be supplied at the abstract relational layer and stored in appropriate structures belonging to this layer. Also parameters for mapping the abstract relational model to a target SQL dialect must be supplied and stored.

Current modeling tools often intermix model properties with mapping information or do not include mapping information at all. If the mapping algorithm of a modeling tool does not allow the user to generate a relational model with the desired properties, manual changes might be necessary at the relational level.

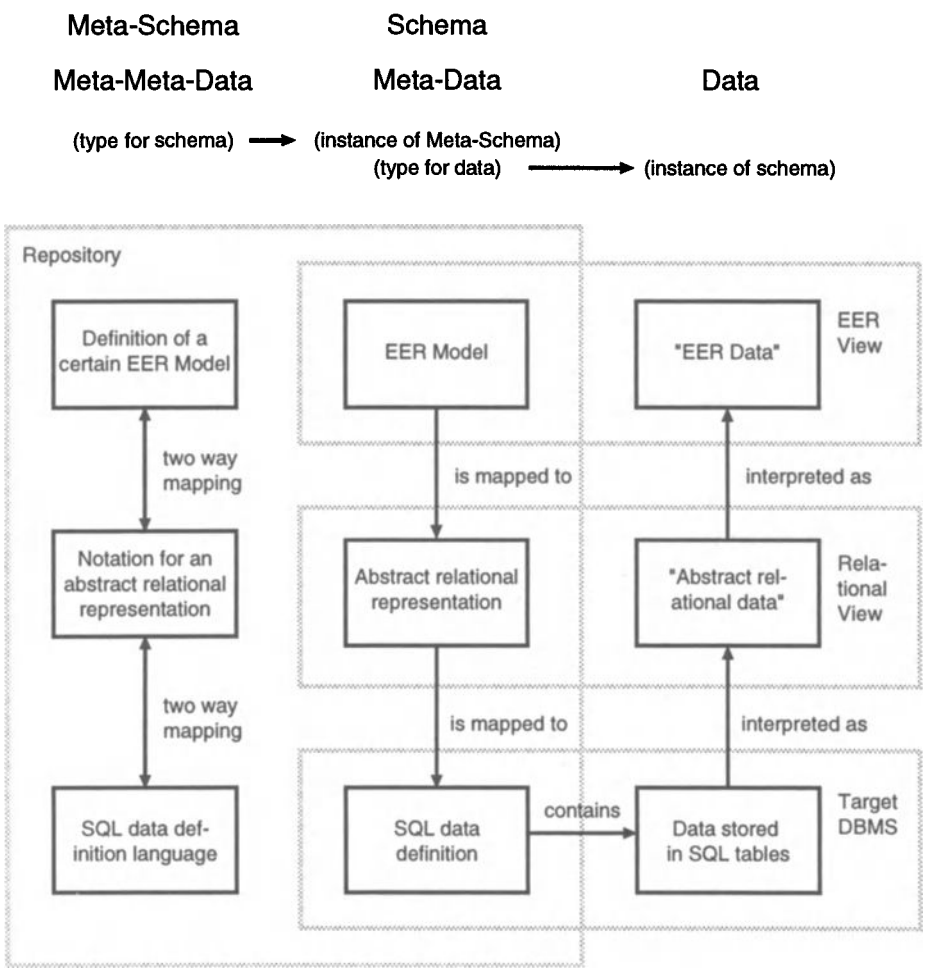


Fig. 1. Meta-Schema, Schema and Data

This will result in a situation where the three layered approach will at the most be used for the initial design, but not during system development.

3 Repository Design

In our opinion, a well designed repository structure is needed to support the storage of all design information including the models and interrelationships between the models. This would allow tight synchronization, and facilitate browsing and the investigation of design implications.

A good test of such a repository would therefore be its use as a basis for visualization of interrelationships within and between meta data and data for an information system. In a broader context such a repository could also act as a unified store for a number of different tools, including schema design tools. Figure 1 depicts the repository content: its structures (left column) and the *type* data it contains (middle column). *Instance* data is not contained in the repository; this resides in the target DBMS (bottom row).

> "Repositories must maintain an evolving set of representations of the same or similar information. [...] Relationships among these representations must be maintained by the repository, so changes to one representation of an object can be propagated to related representations of the same object.", [18, Section 5.3.3],

Interrelationship information in the repository allows the interpretation of data in the target application database at three levels (as depicted in Figure 1): that of the "Target DBMS", that of the "Relational View" and that of the "EER View".

We were looking for standardized repository schemas and have chosen CDIF. As the name suggests it is "a single architecture for exchanging information between CASE tools, and between repositories" which is "vendor-independent" and "method-independent" [8]. It should overcome the problem that no commonly agreed standard repository exists by defining a way of transferring data on the same *subject area* from one tool or repository to another. Case data transfer is not the topic of this project. However, the structures offered by the subject areas *Data Modeling* (DMOD) and *Presentation, Location & Connectivity* (PLAC) were chosen as a basis for a repository structure to store EER and relational models and their graphical representation. The separation between models and their graphical representation supports a modular design. All subject areas are hierarchically structured and make extensive use of inheritance to minimize redundancy. This suggests a direct mapping to a class hierarchy of an object oriented host language. The full range of objects and relationships of DMOD is displayed in Figure 2 and our simplified implemented model is depicted in Figure 3. The associated PLAC design is not depicted.

The DMOD subject area is capable of storing EER- as well as relational models in a single structure. Most objects are used for both layers, some (e.g.

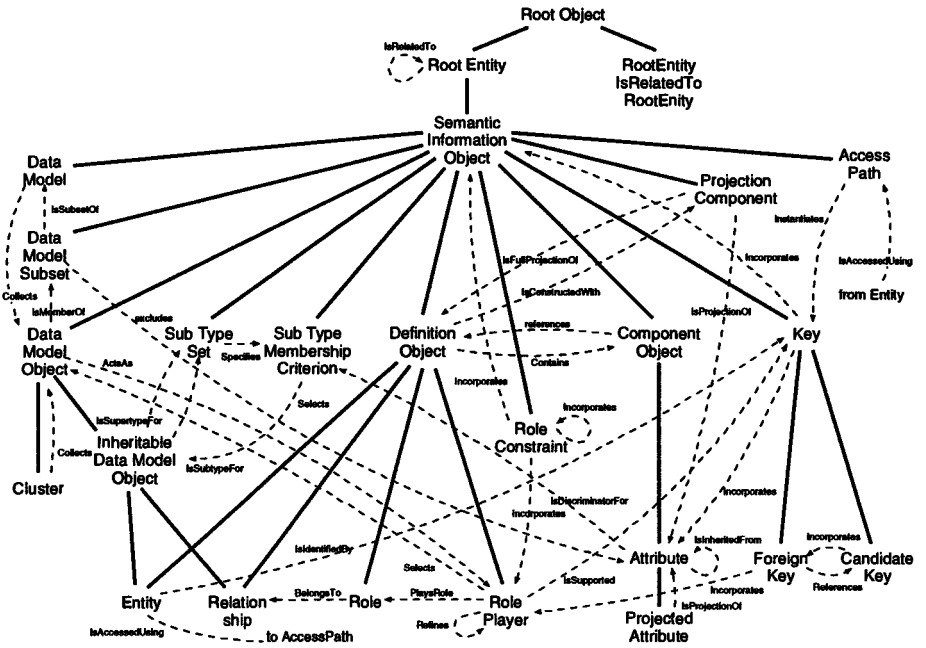


Fig. 2. CDIF-DMOD: overview of objects and relationships

foreign keys or supertype/subtype) only for one. CDIF does not provide structures to relate EER objects to relational objects. We circumvent this problem by extending the subject-area to cope with these correspondences as depicted in Figures 3 and 4. Migration rules for implementing primary and foreign relational keys from EER candidate keys are not included in the model, since we rely on the mapping procedure of the S-Designor modeling tool which does not support them.

In the following we introduce a visualization tool based on the repository, which depicts the different layers shown in Figure 1.

4 Visualization Tool

4.1 Visualization Architecture

Initially it was planned to have a single view for the complete visualization, including data and schema. This would mean, for example, that at the EER level the visualization would include entity instances as well as relationship instances. It would be possible to include this information into a schema only for very limited example models. Scalability is often a problem with such visualization. It is not only a technical or geometric issue of placing graphical elements on a limited two dimensional space but also of comprehensibility for users. This problem led to the overall visualization architecture as laid out in this chapter.

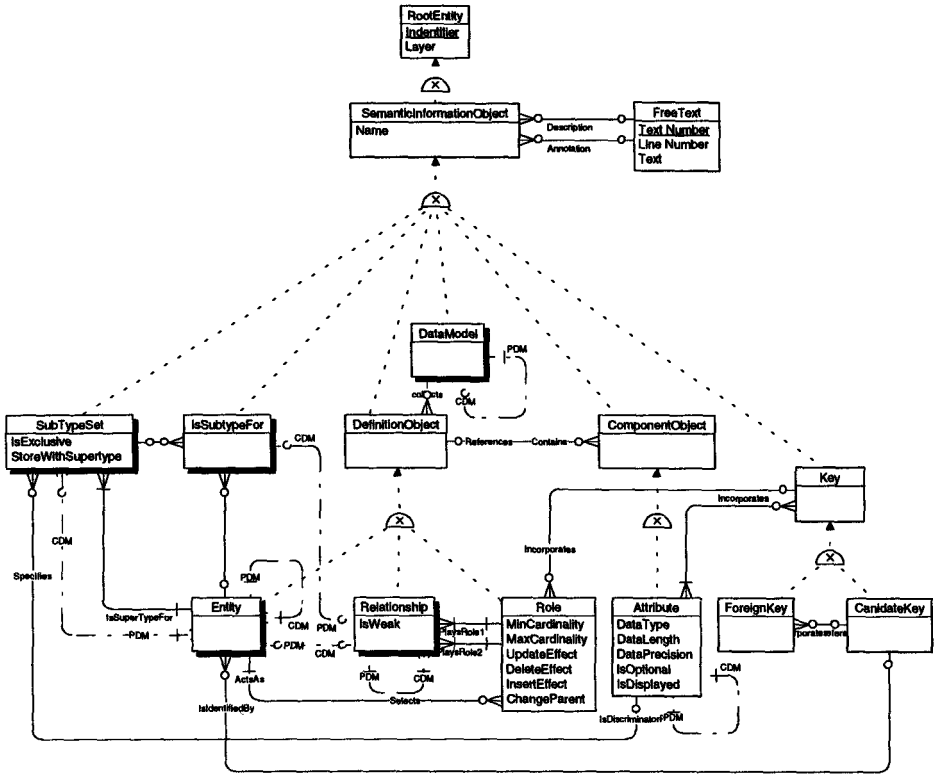


Fig. 3. Conceptual schema of the repository structure

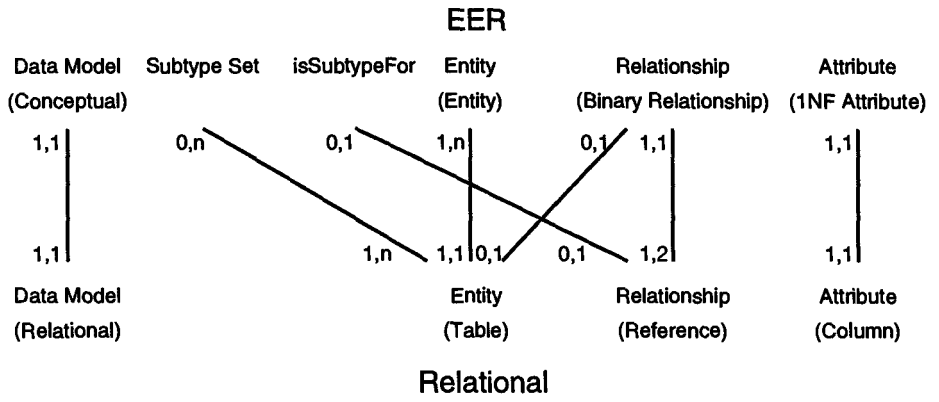


Fig. 4. Mapping-meta-relationships (layer dependent terms in parentheses; numerical cardinalities given (min,max))

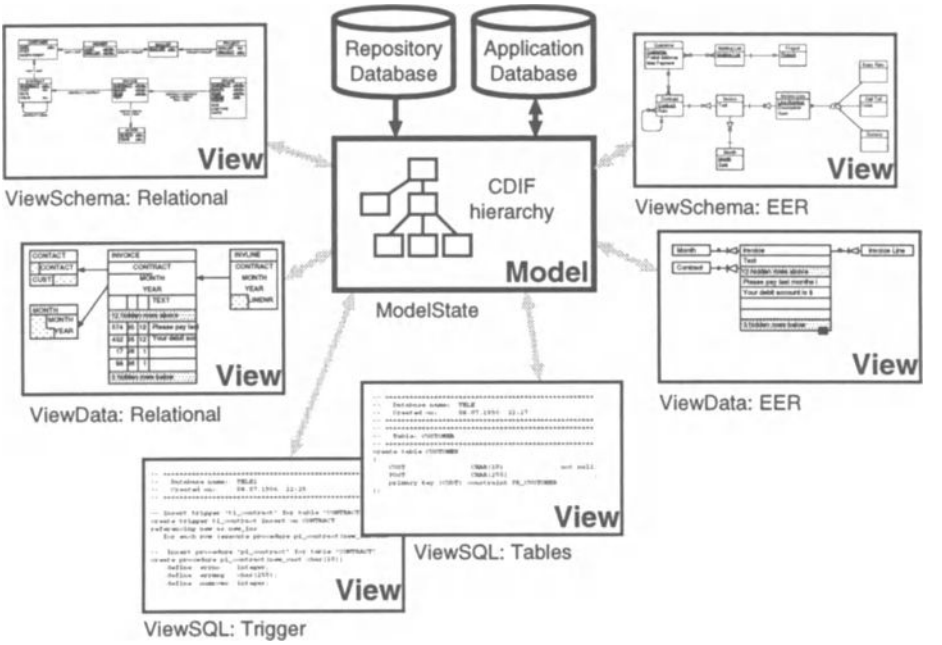


Fig. 5. Model and Views

The basic idea is to split the wealth of displayable information into manageable and comprehensible pieces and at the same time to relate these partial views through visual feedback. Views are deliberately limited to only certain aspects of the whole, as represented by the boxes in Figure 1 (Meta-Schema, Schema, Data combined with EER, relational, SQL). Examples of views appear below.

We make use of the ideas and terminology of the Model-View-Controller (MVC) paradigm which stems from the Smalltalk environment: "The *model*[1] is used to represent the data or knowledge about the application constructed" [2] and the *view* is the output interface, while the *controller* takes inputs from the user and passes them to the *model*. "A major idea in the MVC is the isolation of the model from the view and controller. Because the model normally represents a set of data or knowledge, it has no need to know how the controller or the view operates" [2].

Logically, the model is a server which accepts connections from views and controllers (clients) and communicates with them via a protocol (arrows in Figure 5). The model has direct access to the repository (data model content (DMOD) and its graphical presentation (PLAC)) and the target database, while the views access the repository data only via the connection to the model.

If the user interacts with one view, the result should not only be apparent in this view, but in all open views at the same time. This concerns highlighting

[1] Not to be confused with "data model".

of objects and update visualization. A simple example is a click on an entity in an EER-view and a visual highlighting feedback in all views which display information on the same object. An abstract relational view would highlight the appropriate relation and a SQL-view would highlight the relevant SQL data definition statements. The granularity of highlighting and displaying should be as small as possible, i.e. if a simple attribute should be shown, then only that attribute should be highlighted and not the whole entity or relation.

It should be possible to open new views with a focus on a particular object by clicking on that object in an already existing view. In this way one can browse a model according to the matrix of Figure 1. Relating objects in different views is also of value for managing sub-models with common objects.

4.2 Supported Views

The following short descriptions give an overview of the possible types of view. Firstly, views are introduced which are limited to visualizing the schema (middle column in Figure 1):

Schema: provides a conventional graphical representation of the EER model or relational model without including data. The data for screen layout and textual annotation is taken from the repository (PLAC).

SQL: Since SQL code is generated automatically using templates of the schema design tool it is possible to include markers which allow this view to highlight data definition statements. In this way the SQL code can be browsed by clicking on objects in other views.

Text: "Validation can be done by paraphrasing a conceptual schema in natural language and giving that paraphrase to a user for examination" [4]. The explanations may be automatically derived from the schema semantics or could include textual descriptions. This view reacts to highlighting messages from other windows.

The following views allow visualization and manipulation of instance level data (right column in Figure 1). The schema views above should nevertheless react to messages concerning data manipulation, by using them to highlight relevant schema objects. Different colours can indicate the action performed (for example red=delete, green=insert, blue=update). If a trigger is initiated, its behaviour could be simulated in a single step mode. The **SQL**-view might display a related trigger definition.

Data: This view is most meaningful for visualizing data and update propagation and can be used for EER and relational models. The screen layout is exemplified in Figure 6. Only objects of interest are shown, but browsing allows further schema exploration (upper mouse click in Figure 6). The screen layout can be derived from the schema semantics of the relational layer, since references are directed. Referenced relations appear to the right of referencing relations. The layout of the EER model can be determined with

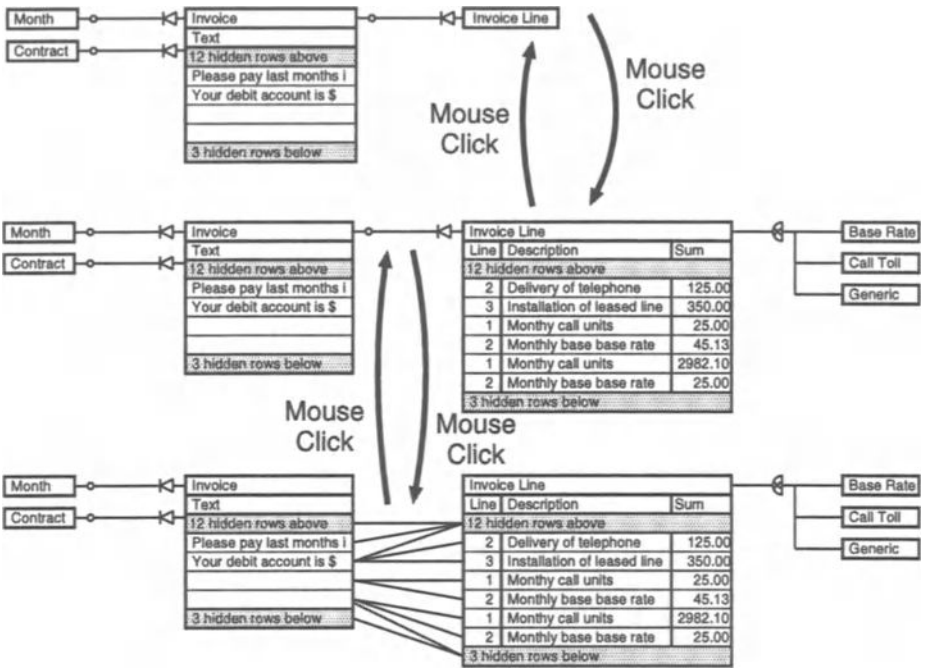


Fig. 6. Data-View example

the aid of the relational representation. Objects may be popped up to show instance level data or popped down to hide it (lower mouse click in Figure 6). Relationship instances as well as data tuples may be graphically inserted, deleted and updated. Submission of data may trigger update propagation, which can be visualized in a single step mode.

Cascade: This passive view is an implementation of the rule trace trees used in [10] and [7]. The first event of a cascade is drawn at the left border as depicted in Figure 7, and all subsequently invoked rules and triggered events are drawn to the right of it, i.e. the flow of time is visualized. Rules may be visually linked to SQL triggers, and events to sets of data tuples.

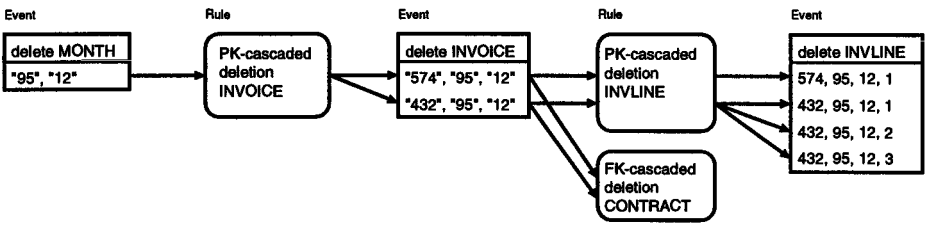


Fig. 7. Sample triggering cascade

Trace: This passive view records the active updates of the database in a textual way comparable with the log file of a DBMS.

Tree: A collapsible tree, as known from file-browsers, is used to access the repository contents in a tabular fashion. It is structured by object type and allows inclusion of instance level data.

4.3 Implementation

The S-Designor database design tool was used to edit an EER-Model and to map it to its relational representation. The tool saves both models, represented as relational tables, in separate plain ASCII-files. The tool was also used to generate tables and triggers for the target DBMS. All files are parsed and loaded into the developed repository. Additionally, the table and trigger SQL-scripts are executed on the target application database.

The visualization tool was implemented using the Java programming language. The tool can be executed within WWW pages[2] assuming that a Java capable browser is used. It becomes a part of the overall presentation. Implemented features are presented next to designed or planned features. The class structure for repository access was completed, but only a selection of views were implemented.

A loader class reads the model via JDBC[3] and instantiates each object as a Java object. Relationships are modeled as object references. Ideally one would use a persistent object oriented programming language based on an object oriented DBMS where "a repository manager would map its object base into labeled directed graphs, where objects are mapped to nodes and relationships are mapped to edges." [1, pp. 710–711]

The model in the MVC sense is a Java class which inherits from the standard class "Observerable" and each view consists of a Java class which inherits from "Observer" to allow the *observerable* object to notify its *observers*. The protocol for communication between views and the model consists of public methods in the model class.

Application data is accessed from the relevant meta-data classes. Update propagation must be simulated by the classes, since we want to execute and visualize update propagation in a step by step fashion. Java provides a rollback mechanism through exceptions to model transactions. We use update propagation properties which are included in the repository and do not consider, for example, SQL-triggers. Visualization could be extended to arbitrary active behaviour if a direct interface to the rule manager of a DBMS existed. This would also allow visualization of updates executed by application programs.

5 Conclusions

In this paper we have described work in the area of repository design for Information Systems development. This work targets tool-independent storage of design

[2] URL http://www.his.se/ida/research/groups/dbtg/demos/griebel/

[3] Java Database Connectivity

information, facilitating transparency within and between stages in the refinement process. The focus of the paper is on a specific facet, namely a repository structure for the multi-level visualization of dependencies in populated models using EER to Relational refinement methods.

Many variations of ER/EER models have been proposed in the literature, and it is not uncommon for a commercial modeling tool to provide support for more than one such variation. For example, both LogicWork's tool ERwin/ERX and Powersoft's tool PowerDesignor support more than one EER dialect [16] [17].

Using a meta-model approach (i.e. a generic ER/EER meta-schema) as exemplified by the system described here, a semantically expressive meta-model could, at least in principle, support any variation of the ER/EER Model.

Populated models are used in many database oriented application development environments. For example, Borland's Delphi environment enables instant access to data kept in an underlying database during the design process. We consider them fundamental in improving feedback in the design phase for behavioural models.

The unification of design data within a repository facilitates the development of generic tools, with presentation levels tailored to organizational standards. The tool presented here is a visualization tool, allowing multiple, interlinked views of populated models to be simultaneously displayed. The tool is itself of interest in exploring useful facilities for visual browsing of designs. However, it primarily fulfils the role of a practical test of the repository design.

References

1. Philip A. Bernstein and Umeshwar Dayal. An Overview of Repository Technology. In *Proceedings of the 20th VLDB Conference*, pages 705–713. VLDB Conference, 1994.
2. John R. Bourne. *Object-Oriented Engineering — Building Engineering Systems Using Smalltalk-80.* Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992.
3. S. Brinkkemper. Integrating diagrams in CASE tools through modelling transparency. *Information and Software Technology*, 35(2):101–105, February 1993.
4. Janis A. Bubenko and Benkt Wangler. Research directions in conceptual specification development. Technical Report 91-024-DSV, SYSLAB, Department of Computer Science and Systems Science Stockholm University and SISU — Swedish Institute for Systems Development, November 1991.
5. J. A. Bubenko jr, R. Dahl, M. R. Gustafsson, C. Nellborn, and W. Song. Computer Support for Enterprise Modelling and Requirements Acquisition. Technical Report Deliverable 3-1-3-R1 Part B, Swedish Institute for Systems Development (SISU), November 1992.
6. Janis A. Bubenko jr. Extending the Scope of Information Modeling. In A. Olivé, editor, *Fourth International Workshop on the Deductive Approach to Information Systems and Databases*, pages 73–97. Lloret de Mar, Costa Brava, September 20–22 1993.
7. S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A visualization and explanation tool for debugging eca rules in active databases. Technical Report UF-CIS-TR-95-028, University of Florida, November 1995.

8. Johannes Ernst. Introd. to CDIF, 1997. URL http://www.cdif.org/intro.html.
9. Christian Fahrner and Gottfried Vossen. A Survey of Database Design Transformations Based on the Entity-Relationship Model. *Data & Knowledge Engineering*, 15(3):213–250, 1995.
10. Thomas Fors. Visualization of Rule Behavior in Active Databases. In S. Spaccapietra and R. Jain, editors, *Visual information management: Proceedings of the third IFIP 2.6 working conference on visual database systems*, pages 215–231. Chapman & Hall, 1995.
11. Gerrit Griebel. Repository Support for Visualization in Relational Databases. Master's thesis, University of Skövde, September 1996.
12. Björn Lundell. Penning a Methodology: A Classroom example used as a framework for reasoning about Information Systems Development. In N. Jayaratna, T. Wood-Harper, and B. Fitzgerald, editors, *Proceedings of the 5th BCS Conference on "Training and Education of Methodology Practitioners and Researchers"*. The British Computer Society: Specialist Group on Information Systems Methodologies, Preston, UK, August 1997.
13. Björn Lundell and Brian Lings. Expressiveness within Enhanced Models: An Infological Perspective. In S. W. Liddle, editor, *Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling*. UCLA, Los Angeles, California, November 1997. URL http://osm7.cs.byu.edu/ER97/workshop4/.
14. Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993.
15. Powersoft. *S-Designor DataArchitect 5.0 User's Guide*. Powersoft Corp., 1996.
16. Robin Schumacher. ERwin/ERX 3.0. *DBMS*, 10(11):31–32, October 1997. URL http://www.dbmsmag.com/9710d08.html.
17. Robin Schumacher. PowerDesigner 6.0. *DBMS*, 10(11):34,36,49–50, October 1997. URL http://www.dbmsmag.com/9710d09.html.
18. Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Research: Achievements and Opportunities Into the 21st Century. *SIGMOD Record*, 25(1):52–63, 1996.
19. Asterio K. Tanaka. *On Conceptual design of Active Databases*. PhD thesis, Georgia Institute of Technology, December 1992.

Observation Consistent Integration of Views of Object Life-Cycles *

Günter Preuner, Michael Schrefl

Department of Business Information Systems
University of Linz, A-4040 Linz, Austria
(preuner | schrefl) @dke.uni-linz.ac.at

Abstract. A commonly followed approach in database design is to collect user views on the database and to develop the conceptual schema of the database by integrating these views.
The design of object-oriented databases involves the design of object behavior next to the design of object structure. Object-oriented design notations usually represent object behavior at two levels of detail: by activities (which correspond to methods at the implementation level) and by object life-cycles which model the dynamics of objects over their lifetime.
This paper discusses the integration of views of object life-cycles that are represented by behavior diagrams, which model the behavior of objects by activities and states corresponding to transitions and places of Petri nets. The presented approach is particularly relevant for the design of business processes, a major application domain of object-oriented database systems.

1 Introduction

The design of an object-oriented database schema involves the definition of the structure and the behavior of objects. Behavior is specified by methods and object life-cycles, which indicate processing states of objects and specify in what order methods may be invoked. Often the database schema is not modeled by a single person, but several user-specific view schemas are defined independently and are integrated later into a single schema.

A major application domain for object-oriented database systems is the business area. There, objects represent business cases and object life-cycles represent business processes. Such as the conceptual schema of a database is often developed from different user views on the database, a business process may be constructed from the views on the business process in different suborganizations.

Consider, for example, a hotel, where the business objects are room reservations: The reception deals with activities check-in and charging services; the booking department considers booking a room and cashing a deposit, etc. Activity check-in will be considered in both views as the guest checks in at the

* This work was partly supported by the EU under ESPRIT-IV WG 22704 ASPIRE (Advanced modeling and SPecification of distributed InfoRmation systEms).

reception and the booking department checks whether the guest arrives at the agreed day.

As each view represents only those aspects of the business process that are relevant for a certain user-group, each view may cover only a subset of the aspects of the business process and each view may represent aspects of the business process at a different level of granularity than another view.

The aim of the integration process is to develop an integrated business process from the given views such that if business objects are processed according to the description of the integrated business process, their processing may be *observed* as a correct processing according to each view of the business process *(observation consistent integration)*.

In [3], we introduced a two-schema architecture for modeling *business processes* and *workflows*, which represent *external* and *internal* business rules, respectively. External business rules restrict the processing of business objects due to natural facts or law (e.g., check-in is always performed before check-out, but payment and use of a room may occur in any order). Internal business rules exist only due to intra-organizational rules and represent the internal "flow of work" (e.g., payment has to be performed *before* check-in if the guest is unknown).

The distinction between business processes and workflows provides for *workflow transparency*, which allows to change the flow of work without compromising the correctness of the business process. In this paper, we treat the integration of views of repetitive and predictable business processes represented by object life-cycles; we do not consider the integration of workflows and omit definition of exception handling and ad-hoc processing of business objects.

View integration differs from integration of heterogeneous autonomous data sources, which is mainly treated in the realm of federated database systems [13]. In the former, there is only *one* object for each real-world business case; this object behaves according to the *integrated business process*. The views do not handle objects of their own, but only *observe* the processing of objects. In the latter, the global schema is an integration of several local schemas where each object of the real world may be represented by a set of local objects, one in each local autonomous database, and by a global object whose properties are derived from the local ones. Despite their difference, there are many similarities between the integration of views and the integration of independent heterogeneous database schemas, e.g., the detection of heterogeneities between the given schemas.

A number of approaches to schema integration consider only the structure of objects (see e.g. [4] for a survey). Some approaches consider methods, too [14, 16], but little work has been done so far concerning the integration of *object life-cycles*. In [5], behavior is modeled in the form of event structures and *conflictfreeness* is defined as a necessary correctness criterion to integrate schemas. Conflictfreeness means that the rules of one schema together with a set of integration assertions do not restrict the models of the other schema. Another approach [6] discusses view integration based on statecharts where views may be connected in some way (e.g., sequentially, parallely, alternatively) to an inte-

grated schema. In a previous paper [9], we discussed the integration of business processes that treat *disjoint sets* of business objects in autonomous databases, where the common behavior can be observed at the global level.

The work presented in this paper goes beyond the approaches of [5] and [6] as we introduce a detailed integration process that may be applied to views and that produces step by step an integrated business process. The integration of views of business processes is different from the integration of local business processes as described in [9] for two reasons: (1) The former concerns view integration, the latter the integration of similar, autonomously executed business processes. (2) The extensions of the given schemas are the same in the former, but disjoint in the latter.

We consider the integration of views of business processes that are modeled using *Object/Behavior Diagrams* (OBD) [7]: The static properties of objects are defined in Object Diagrams, whereas their life cycle is modeled in Behavior Diagrams, which are based on Petri nets [8].

We will define the correctness of the integrated schema with respect to the given view schemas by using results on the *observation consistent specialization of object life-cycles,* which has been treated in several approaches based on Petri nets [10, 11, 15]. Our work goes beyond these approaches as not a *single* object life-cycle is specialized but a common subtype of a *set of object life-cycles* (i.e., the views) has to be defined where correspondences and heterogeneities between the views have to be determined. Specialization of object life-cycles comprises *refinement*, where activities and states are refined to a finer level of granularity, and *extension*, where activities and states are added.

The remainder of this paper is structured as follows: Sect. 2 gives a brief introduction of behavior diagrams and the specialization of behavior diagrams, Sect. 3 gives an overview of heterogeneities between the views and an overview of the integration process, Sect. 4 describes each step of the integration process in detail, and Sect. 5 concludes the work.

2 Object/Behavior Diagrams

In this section, we briefly introduce *Object/Behavior Diagrams (OBD)*, which have been originally presented as a graphical notation for the object-oriented design of databases [7] and have been later extended for the modeling of business processes [3]. We omit the description of object diagrams and concentrate on behavior diagrams with arc-labels and their specialization; for details the reader is referred to [10, 11].

2.1 Behavior Diagrams

Behavior diagrams are based on Petri nets and consist of activities, states, and arcs. Activities correspond to transitions in Petri nets and represent work performed with objects, states correspond to places in Petri nets and show where an object actually resides. Each instance of an object type is represented by a

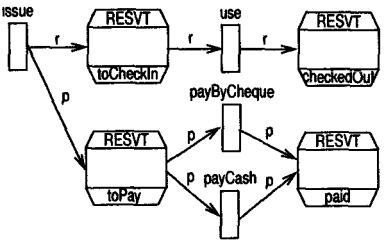


Fig. 1. Behavior diagram of RESVT

unique token, the object identifier, which may reside in one or several states. An activity may be invoked on an object if all prestates are marked by this object. When the execution is completed the object is inserted into all poststates of the activity. In difference to Petri nets, we assume that activities may take some time. During the execution of an activity on some object, the object resides in an implicit activity state named after the activity.

Example 1. Fig. 1 shows a behavior diagram of object type RESVT. Activities are depicted by rectangles, and states are depicted by rectangles with a trapezium at the top and the bottom. Ignore the labels associated with arcs for the moment. Activity issue creates a new reservation object. After completion of this activity, the object resides in states toCheckIn and toPay, where activities use and payByCheque or payCash may be started.

Behavior diagrams may be *labeled.* The idea of labeling is taken from the processing of business processes by paper work where different actors work on different carbon copies of a business form. In this analogy, a label corresponds to a particular carbon copy. The labels of an arc (state, or activity) indicate which copies of a form flow along an arc, (reside in some state, or are processed by an activity, resp.). Notice, however, that *there exist no actual copies of an object*, but several activities and states may refer at the same time to the same object by its object identifier. Labels have been introduced in [11] to facilitate the check whether a behavior diagram constitutes a consistent refinement of another behavior diagram. There, labels are used to ensure that if an object leaves one state of a subdiagram (that constitutes the refinement of an abstract state), it leaves the subdiagram entirely.

Example 2. The business process shown in Fig. 1 uses two labels p and r corresponding to the payment and the registration copy of the reservation form.

Formally, *labeled behavior diagrams* are defined in Def. 1. For a formal definition of *unlabeled* behavior diagrams see [10].

Definition 1. *A* labeled behavior diagram *(LBD) B of an object-type, where $B = (S, T, F, L, l)$, consists of a set of states $S \neq \emptyset$, a set of activities $T \neq \emptyset$, $S \cap T = \emptyset$, a set of arcs $F \subseteq (S \times T) \cup (T \times S)$, such that $\forall t \in T : (\exists s \in S : (s,t) \in F) \wedge (\exists s \in S : (t,s) \in F)$ and $\forall s \in S : (\exists t \in T : (s,t) \in F) \vee (\exists t \in T : (t,s) \in F)$.*

L is a set of labels. The labeling function $l : F \to 2^L \setminus \{\emptyset\}$ (2^X denotes the power set of X) assigns a non-empty set of labels to each arc in F. States, activities, and labels are refered to as elements. *There is a set of* initial states $A \subset S$ *such that* $\nexists (t,s) \in F : s \in A$ *and there exists a set of* final states $\Omega \subset S$ *such that* $\nexists (s,t) \in F : s \in \Omega$. *Initial states are usually omitted in the graphical representation.*

Labeled behavior diagrams must fulfill the following labeling properties: *(1) label preservation* (for each activity, the union of the labels of its incoming arcs is equal to the union of the labels of its outgoing arcs), *(2) unique label distribution* (all incoming arcs as well as all outgoing arcs of an activity have disjoint sets of labels), and *(3) common label distribution* (for each state, all its incident arcs carry the same labels).

States and activities are labeled, too, where the set of labels of a state or activity, denoted as λ, is the union of the set of labels of its incident arcs.

Each label appears in exactly one initial state, i.e., $(\bigcup_{\alpha \in A} \lambda(\alpha)) = L$ and $\forall s_1, s_2 \in A : s_1 \neq s_2 \Rightarrow \lambda(s_1) \cap \lambda(s_2) = \emptyset$. An initial state represents the period of time when a certain label has not yet been created for an object. If an object identifier resides only in initial states, it does not yet refer to an object.

At any time, an object resides in a non-empty set of states, its *life cycle state.*

Definition 2. *A* life cycle state *(LCS) σ of an object is a subset of $(S \cup T) \times L$. The initial LCS is $\sigma_A = \{(\alpha, x) | \alpha \in A \wedge x \in \lambda(\alpha)\}$. An LCS σ is* final *if $\forall (e,x) \in \sigma : e \in \Omega$.* Note: *For* unlabeled *behavior diagrams, an LCS is defined as $\sigma \subset (S \cup T)$.*

As the execution of activities may take some time and an object resides in an activity state during execution, we distinguish *start* and *completion* of an activity and define the change of the LCS of an object as follows (we use the notation ${}^\bullet t$ and $t^\bullet$ for the set of pre- and poststates of t, respectively).

Definition 3. *An activity $t \in T$ may be started on an LCS σ, if $\forall s \in {}^\bullet t, x \in l(s,t) : (s,x) \in \sigma$, and its start yields $\sigma' = (\sigma \setminus \{(s,x) | s \in {}^\bullet t \wedge x \in l(s,t)\}) \cup \{(t,x) | \exists s \in {}^\bullet t : x \in l(s,t)\}$. An activity t may be completed on an LCS σ, if $\exists x \in L : (t,x) \in \sigma$, and its completion yields $\sigma' = (\sigma \setminus \{(t,x) | \exists s \in t^\bullet : x \in l(t,s)\}) \cup \{(s,x) | s \in t^\bullet \wedge x \in l(t,s)\}$.*

A *life cycle occurrence* is defined as the sequence of life cycle states of a certain object:

Definition 4. *A* life cycle occurrence (LCO) *γ of an object is a sequence of LCSs $\gamma = [\sigma_1, \ldots, \sigma_n]$, such that $\sigma_1 = \sigma_A$, and for $i = 1, \ldots, n-1$ either $\sigma_i = \sigma_{i+1}$, or $\exists t \in T$ such that either t can be started on σ_i and the start of t yields σ_{i+1} or $\exists x \in L : (t,x) \in \sigma_i$ and the completion of t yields σ_{i+1}.*

Example 3. Consider the behavior diagram shown in Fig. 1. A possible life cycle occurrence of a reservation object is [{(α, p), (α, r)}, {(issue, p), (issue, r)}, {(toPay, p), (toCheckIn, r)}, {(toPay, p), (use, r)}, {(toPay, p), (checkedOut, r)}, {(payCash, p), (checkedOut, r)}, {(paid, p), (checkedOut, r)}].

2.2 Specialization of Behavior Diagrams

The behavior diagram of an object type may be specialized in two ways: by *refinement*, i.e., by decomposing states and activities into subdiagrams and labels into sublabels, or by *extension*, i.e., by adding states, activities, and labels. *Observation consistency* as a correctness criterion for specialization guarantees that any life cycle occurrence of a subtype is observable as a life cycle occurrence of the supertype if extended elements are ignored and refined elements are considered unrefined. Observation consistency allows *"parallel extension"* but not *"alternative extension"*, i.e., an activity that is added in the behavior diagram of a subtype may not consume from or produce into a state that the subtype inherits from the behavior diagram of the supertype [10]. Observation consistent extension requires only partial inheritance of activities and states: "alternatives" modeled in the behavior diagram of a supertype may be omitted in the behavior diagram of a subtype (cf. [10]). [1]

We use a total specialization function $h : S' \cup T' \cup L' \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ to represent the correspondences between a more special behavior diagram $B' = (S', T', F', L', l')$ and a behavior diagram $B = (S, T, F, L, l)$. Inheritance without change, refinement, extension, and elimination are expressed by h as follows: If an element e is not changed, then $\exists e' \in S' \cup T' \cup L' : h(e') = e \wedge \forall e'' \in S' \cup T' \cup L', e'' \neq e' : h(e'') \neq h(e')$. If an element e in B is refined to a set of elements E ($|E| > 1$), then $\forall e' \in E : h(e') = e$. If a set of elements E is added in B', then $\forall e \in E : h(e) = \varepsilon$. If a set of states and activities $E \subseteq S \cup T$ is removed from B in B', then $\forall e \in E \nexists e' \in S' \cup T' \cup L' : h(e') = e$.

For the definition of *observation consistent specialization*, we need to define the *generalization of a life cycle state and a life cycle occurrence.*

Definition 5. *A* generalization of a life cycle state σ' *of an LBD* B' *of object type* O' *to object type* O *with LBD* B, *denoted as* $\sigma'/_O$, *is defined as* $\sigma'/_O \subseteq (S \cup T) \times L$, *where* $\forall e \in S \cup T, x \in L : ((e, x) \in \sigma'/_O \Leftrightarrow \exists e' \in S' \cup T', x' \in L' : h(e') = e \wedge h(x') = x \wedge (e', x') \in \sigma')$.

Definition 6. *A* generalization of a life cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ *of an LBD* B' *of object type* O' *to object type* O *is defined as* $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.

Definition 7. *An LBD* B' *of object type* O' *is an* observation consistent specialization *of an LBD* B *of object type* O *if and only if for any possible LCO* γ' *of* B' *holds:* $\gamma'/_O$ *is an LCO of* B.

Example 4. The LBD of RES-R shown in Fig. 2 is an observation consistent specialization of the LBD of RESVT in Fig. 1. To avoid overloading of Fig. 2, the unrefined elements inherited from RESVT are not depicted. The symbol with

[1] Omitting states and activities of a supertype in a subtype seems to be counter-intuitive at first. Yet, a more in depth analysis (cf. [10]) reveals that partial inheritance is coherent with a co-variant specialization of method preconditions followed in conceptual specification languages.

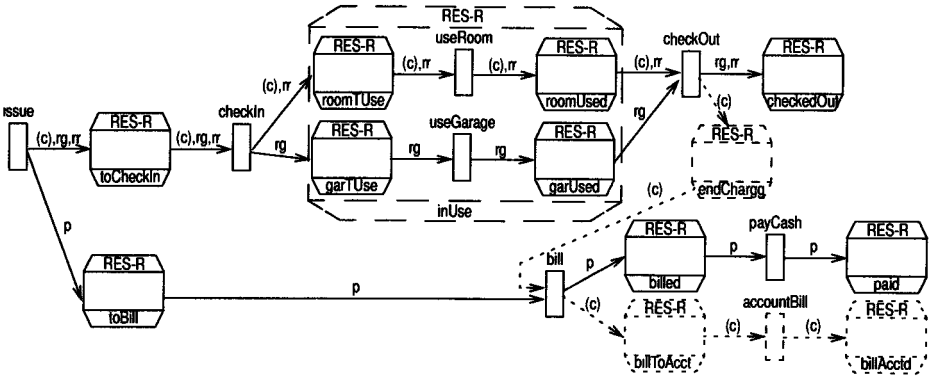


Fig. 2. Business process of RES-R

dashed borders, the brackets around label c, and the fact that some symbols are depicted with dotted lines are explained in the next section.

Activity use has been refined to a subnet consisting of activities checkIn, useRoom, useGarage, and checkOut and several states. State toPay has been refined to toBill, bill, and billed. Label r has been decomposed into sublabels rr (registration for a room) and rg (registration for a parking lot in the garage). Activity accountBill, states endChargg, billToAcct, and billAcctd, as well as label c have been added by extension. Activity payByCheque has been omitted.

A set of sufficient and necessary rules has been introduced in [11] to check for *observation consistent extension* and *refinement* of LBDs. These rules can be applied to check for *observation consistent specialization* of LBDs, too, if specialization is considered as a concatenation of refinement and extension.

3 Integration of Views of Business Processes

In this section, we describe how to integrate *two* views of a business process. To integrate more than two views, a *binary integration strategy* [1] may be used, which integrates each view with an intermediate integrated schema. The views may differ with respect to several kinds of heterogeneities. We will identify these heterogeneities first; then we will describe the integration process.

3.1 Heterogeneities

Some of the heterogeneities between different views of business processes resemble the heterogeneities identified for federated databases [12]. Others, especially the second and last one described below are unique for business processes: *(1) Model conflicts:* The views may be represented in different models (e.g., statecharts vs. Petri nets). *(2) Inclusion of workflow aspects:* The views may represent workflows rather than business processes (cf. discussion in the introduction). *(3) Missing states and labels:* States and labels that are modeled in

one view may not be modeled explicitly in the other view, but may exist there only implicitly. *(4) Naming conflicts:* Corresponding elements may carry different names *(synonyms)* or different elements may carry the same names *(homonyms)*. *(5) Granularity conflicts:* A single state or activity in one view may correspond to a subnet consisting of several activities and states in the other view, or two subnets may correspond to each other. *(6) View-specific alternatives:* One view may include an alternative that is not modeled in the other view; i.e., an activity that consumes from or produces into a state that has a corresponding state in the other view does not have a corresponding activity in the other view. *(7) View-specific aspects:* One view may consider aspects represented by labels that are not modeled in the other view, i.e., a label in one view does not correspond to any label in the other view. *(8) Different time periods:* The views may represent different periods of the lifetime of business objects. This heterogeneity may be considered as a special kind of granularity conflict, in which the initial or final states of one view correspond to subnets in the other view.

3.2 Overview of the Integration Process

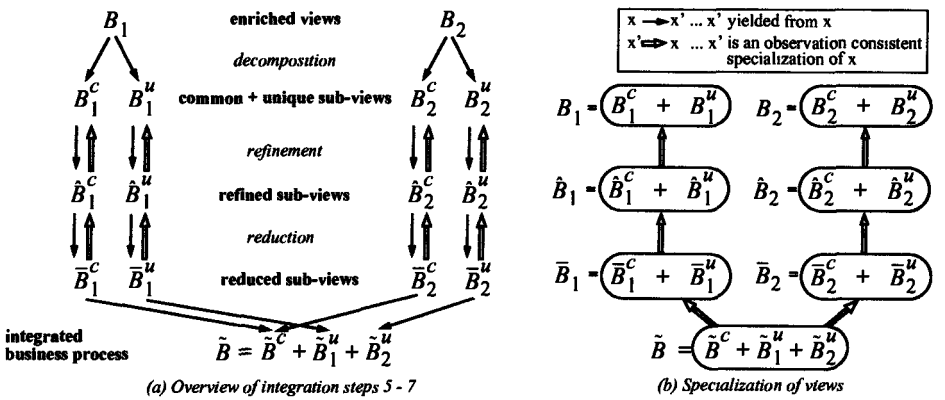
The integrated business process has to be constructed from the given views in a way such that the processing of business objects according to the integrated business process may be observed as correct processing of the business objects according to each given view, i.e., the integrated business process must be an observation consistent specialization of each given view.

As views do not administrate information of their own about business objects, all information that is visible in the view must be derived from the integrated business process. Therefore, the integrated business process must include all information specified in either one of the views, except view-specific alternatives, which must be omitted to provide for an observation consistent specialization, since performing a view-specific alternative on a business object could not be observed in the view that does not include this alternative.

If corresponding elements are defined in the views at different levels of granularity, the most refined definition is taken into the integrated schema to provide the most detailed information possible.

Before the views can be integrated according to the rules of observation consistent specialization, some preparatory steps (steps 1–3) are performed on the original views, thereby yielding *enriched views*. Then, the integrated business process is defined by *refinement* and subsequent *extension* of the enriched views. Some steps are comparable to the steps followed in several approaches of database integration (cf. [2, 13]). In the remainder of this section, we give an overview of the integration process. The steps of the integration process will be treated in detail together with examples in the next section.

1. *Schema translation:* If the given views are modeled using different models, they are translated to a canonical model, thereby resolving *model conflicts*.
2. *Extraction of business processes from workflows:* Heterogeneities due to *inclusion of workflow aspects* are resolved in this phase as only elements concerning the business process are considered.



(a) Overview of integration steps 5 - 7 (b) Specialization of views

Fig. 3. Integration process by observation consistent specialization

3. *Schema enrichment:* Information that exists only implicitly in a given view is made explicit. Thereby, *missing states and labels* are defined.
   Labels should help the system integrator to determine common aspects of the given views and to check for correct refinements of activities and states. Labels can be determined according to the analogy with paper forms introduced in the previous section. Notice that this analogy should only be a hint to find labels as no real paper copies have to exist. A redundant state has to be inserted if it represents the processing state of a certain label but has not been modeled so far in the view.
4. *Identification of correspondences:* Correspondences between activities, states, and labels of the enriched views are determined. Elements of a view that have corresponding elements in the other view are called *common,* all other elements are called *unique.*
5. *View decomposition:* Common elements and view-specific aspects are integrated in two separate steps. To determine the sub-views to be considered in these steps, each *enriched view* B_i (for $i \in \{1,2\}$) is decomposed into (1) a *common sub-view* B_i^c, which contains the restriction of the view to all common labels and all activities, states, and arcs that are labeled with common labels, and (2) a *unique sub-view* B_i^u, which consists of all unique labels and all activities, states, and arcs that are labeled with unique labels (cf. Fig. 3 (a)). These sub-views may be overlapping, as an activity, state or arc may be labeled with common *and* unique labels. Notice that by decomposing an enriched view into a common and a unique sub-view no information is added or removed.
6. *Integration of common elements:* The common sub-views B_1^c and B_2^c are integrated to the *common view* $\tilde{B}^c$. Thereby, *granularity conflicts*, heterogeneities due to *view-specific alternatives,* and *naming conflicts* are resolved. To resolve granularity conflicts, the common sub-views B_1^c and B_2^c are refined to a common level of granularity according to the rules of observation consistent refinement (cf. [11]) yielding *common refined sub-views* $\hat{B}_1^c$ and $\hat{B}_2^c$. To resolve heterogeneities due to view-specific alternatives, alternatives

with common labels in one view that are not modeled in the other view are omitted from $\hat{B}_1^c$ and $\hat{B}_2^c$, whereby the rules of observation consistent extension are obeyed because of the rule of *partial inheritance* (cf. Sect. 2.2). The resulting reduced sub-views $\bar{B}_1^c$ and $\bar{B}_2^c$ are identical up to renaming. To resolve naming conflicts, a common view $\tilde{B}^c$ that is isomorphic to $\bar{B}_1^c$ and $\bar{B}_2^c$ is defined, where the elements of $\tilde{B}^c$ have different "object identities" than their counterpart elements of $\bar{B}_1^c$ and $\bar{B}_2^c$ and possibly different names.

7. *Integration of view-specific aspects:* For each *unique sub-view* B_i^u, activities and states that are included in the unique sub-view B_i^u as well as in the common sub-view B_i^c and that are refined in $\hat{B}_i^c$, are refined in B_i^u, too, yielding a *refined unique sub-view* $\hat{B}_i^u$.
   Then for each $\hat{B}_i^u$, view-specific alternatives that have been omitted in the reduced common sub-view $\bar{B}_i^c$ are omitted from the refined unique sub-view $\hat{B}_i^u$, too, yielding a *reduced unique sub-view* $\bar{B}_i^u$. Naming conflicts (homonyms) between $\bar{B}_1^u$ and $\bar{B}_2^u$ are resolved — as described above for the common sub-views — yielding $\tilde{B}_1^u$ and $\tilde{B}_2^u$. Overlaps between B_i^c and B_i^u are taken into account in the definition of the object-identities of $\tilde{B}_i^u$ in that an element in $\tilde{B}^c$ and an element in $\tilde{B}_i^u$ that have the same counterpart in B_i receive the same "object identity".
   Heterogeneities due to *view-specific aspects* are resolved in that the integrated business process $\tilde{B}$ is defined as composition $\tilde{B} := \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ (cf. Def. 8).

Definition 8. *The* composition of behavior diagrams $B_a = (S_a, T_a, F_a, L_a, l_a)$ *and* $B_b = (S_a, T_b, F_b, L_b, l_b)$ *(denoted* $B_a + B_b$*) yields a behavior diagram* $B_c = (S_c, T_c, F_c, L_c, l_c)$, *where* $S_c = S_a \cup S_b, T_c = T_a \cup T_b, F_c = F_a \cup F_b, L_c = L_a \cup L_b, l_c = \{(f, X) | f \in F_c \wedge X = l_a(f) \cup l_b(f)\}$.

As the common view B_i^c and the unique view B_i^u are refined the same way and as $\hat{B}_i^c$ is an observation consistent refinement of B_i^c, and as $\hat{B}_i^u$ is an observation consistent refinement of B_i^u, it holds that $\hat{B}_i = \hat{B}_i^c + \hat{B}_i^u$ is an observation consistent refinement of B_i (cf. Fig. 3 (b)). Since $\bar{B}_i^c$ is an observation consistent extension of $\hat{B}_i^c$, and since $\bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i^u$, it holds that $\bar{B}_i = \bar{B}_i^c + \bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i$. As the common sub-views $\bar{B}_1^c$ and $\bar{B}_2^c$ are isomorphic up to renaming, as the sets of labels of the common sub-views are disjoint with the set of labels of each unique subview $\tilde{B}_1^u$ and $\tilde{B}_2^u$, and as the labels of the unique sub-views are disjoint, too, it holds that the integrated business process $\tilde{B} = \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ is an observation consistent specialization of B_1 and B_2.

4 The Steps of the Integration Process

In this section, we will present the steps of integration process in more detail, illustrated by an example described below. Notice that, although the steps are described sequentially, there may be several iterations, especially between steps 3 and 7, as lacking redundant states may be recognized only during specialization.

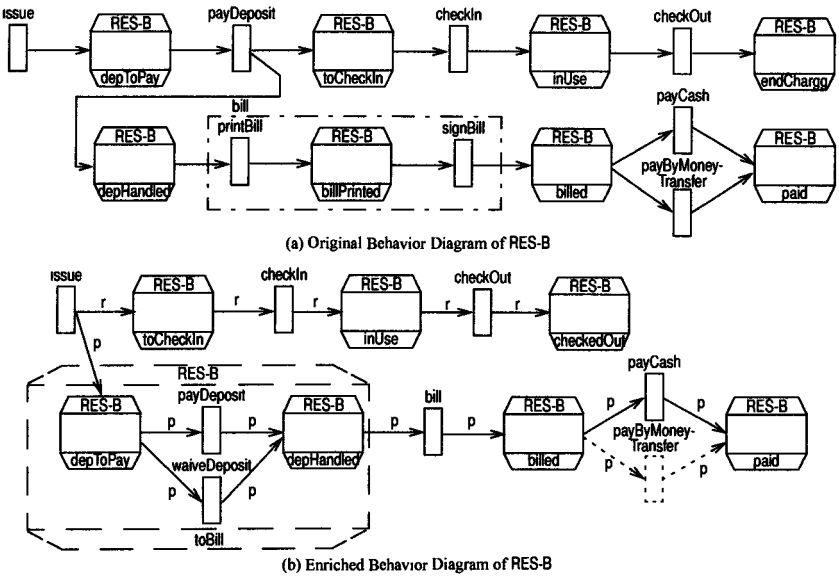


Fig. 4. Behavior Diagrams of RES-B

Example 5. Two views of treating room reservations have to be integrated to a single business process for a hotel. The views consider handling of room reservations from the viewpoints of the reception and of the booking department (cf. Figs. 2 and 4, respectively). The integration steps *extraction of business processes from workflows* and *schema enrichment* will be explained only with RES-B.

4.1 Schema Translation

The translation step is out of the scope of this paper. Herein, we assume that the given view schemas have been translated to unlabeled Behavior Diagrams.

4.2 Extraction of Business Processes from Workflows

The business process is extracted from each view schema that represents a workflow rather than a business process: (a) order dependencies between activities that reflect the intended flow of work rather than the business process itself are omitted; (b) alternative activities that could be executed but are not supported in the considered workflow are included in the view; (c) exact instructions on how a certain unit of work has to be performed by an agent are replaced by the overall business process activity. By omitting restrictions due to internal business rules, contradictions between the views disappear.

Example 6. Consider the view in Fig. 4 (a). Due to an internal business rule a deposit has to be paid before check-in, although, by law, a deposit is not necessary and — in case that it is required — need not necessarily be charged *before*

check-in. Further, activities printBill and signBill and state billPrinted, which represent an internal instruction on how billing is performed by a clerk are replaced by the overall activity bill.

4.3 Schema Enrichment

Information that is not modeled explicitly, but exists only implicitly, is made explicit. Particularly, labels are defined and redundant states may be inserted, i.e., an unlabeled behavior diagram $\breve{B}_i = (\breve{S}_i, \breve{T}_i, \breve{F}_i)$ is transformed into a labeled behavior diagram $B_i = (S_i, T_i, F_i, L_i, l_i)$, where $\breve{S}_i \subseteq S_i, \breve{T}_i = T_i, \breve{F}_i \subseteq F_i$.

Redundant states must not restrict the possible sequences of activity invocations, i.e., the following condition must hold: *For any LCO* $\breve{\gamma} = [\breve{\sigma}_1, \ldots, \breve{\sigma}_n]$ *of* $\breve{B}_i$*, a labeled LCO* $\gamma = [\sigma_1, \ldots, \sigma_n]$ *of* B_i *has to exist such that for* $1 \leq j \leq n$ *it holds that:* $\forall \breve{s} \in \breve{\sigma}_j \exists (\breve{s}, x) \in \sigma_j$.

Example 7. In view RES-B (cf. Fig. 4), labels p and r (representing the aspects *payment* and *registration*, respectively) are introduced, yielding the enriched view shown in Fig. 4 (b).

Assume that the *unlabeled* view RES-R (cf. Fig. 2) did not contain states checkedOut and toBill. Then, these redundant states are inserted during schema enrichment as they represent processing states of labels rg, rr, and p.

4.4 Identification of Correspondences

An activity, state, or label in one view may correspond to an activitiy, state, or label in the other view, either at the same or at different levels of granularity. Thus, we distinguish the following types of correspondences: *(1) Equivalence:* Two activities, states, or labels are equivalent if they correspond to each other. *(2) Inclusion:* A state or activity in one view corresponds to a subnet of activities and states in the other view. Similarly, a single label in one view may correspond to a set of labels in the other view. *(3) No correspondence:* An element in one view corresponds to no element in the other view.

Example 8. In our example, several activities, states, and labels are considered equivalent between the views (cf. Figs. 2 and 4 (b)). For better readability, they carry the same names. E.g., activities issue, states checkedOut, and labels p are equivalent. There exists an inclusion relationship between state inUse in view RES-B and the subnet consisting of roomTUse, garToUse, roomUsed, garUsed, useRoom, and useGarage in RES-R. Further, label r in RES-B corresponds to the set of labels rg and rr in RES-R. Common activities and states are depicted with solid borders, common labels are depicted without brackets, unique activities and states are depicted with dotted borders, unique labels are enclosed in brackets. Subnets in one view involved in an inclusion with a single state e in the other view are visualized by a symbol with dashed borders carrying the name of e.

$\Theta_S^E \subseteq S_1 \times S_2$	$\Theta_T^E \subseteq T_1 \times T_2$	$\Theta_L^E \subseteq L_1 \times L_2$
$\Theta_S^{I_1} \subseteq S_1 \times 2^{S_2 \cup T_2}$	$\Theta_T^{I_1} \subseteq T_1 \times 2^{S_2 \cup T_2}$	$\Theta_L^{I_1} \subseteq L_1 \times 2^{L_2}$
$\Theta_S^{I_2} \subseteq 2^{S_1 \cup T_1} \times S_2$	$\Theta_T^{I_2} \subseteq 2^{S_1 \cup T_1} \times T_2$	$\Theta_L^{I_2} \subseteq 2^{L_1} \times L_2$
$\Theta_S := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_S^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_S^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_S^{I_2}\}$		
$\Theta_T := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_T^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_T^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_T^{I_2}\}$		
$\Theta_L := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_L^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_L^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_L^{I_2}\}$		

Table 1. Correspondence relations

Subnet correspondences, i.e., the fact that two subnets correspond to each other, cannot occur if object life-cycles of two object types modeling the same set of objects are integrated. A subnet correspondence means that an abstract activity or an abstract state is refined differently in two views but an object can be treated only in one way in the integrated schema. Supposed subnet correspondences indicate design errors such as: (1) At least one subnet does not model a pure business process, but considers internal business rules. (2) The subnets consider different aspects (i.e., different labels), and thus, they do not correspond to each other at all. (3) A single element in one subnet corresponds to a single element or a set of elements in the other subnet, i.e., correspondences have not been detected at the finest level of granularity.

We require that correspondences are not overlapping, i.e., that each element e in a view corresponds to at most one element or one subnet E in the other view. Overlapping correspondences may occur if activities and states within the given views have been insufficiently refined such that there are aspects that are defined at an abstract level in both views. In such a case, the views have to be refined such that for each element there is a most-refined representation in at least one view.

For the remainder of the integration process, we define the correspondence relation $\Theta := \Theta_S \cup \Theta_T \cup \Theta_L$ for the enriched views B_1 and B_2, where $B_i = (S_i, T_i, F_i, L_i, l_i)$, for $i \in \{1,2\}$, and where Θ_S, Θ_T, and Θ_L represent correspondences between states, activities, and labels, respectively (cf. Table 1). For each of these correspondence relations, we distinguish between *(1) equivalences* Θ^E (i.e., two basic elements correspond to each other), *(2) left inclusions* Θ^{I_1} (i.e., a single element in B_1 corresponds to a subnet in B_2), and *(3) right inclusions* Θ^{I_2} (i.e., a single element in B_2 corresponds to a subnet in B_1).

As overlapping correspondences are not allowed, no element may appear in more than one correspondence relation, i.e., $\forall (E_1', E_2') \in \Theta, (E_1'', E_2'') \in \Theta$: $(E_1', E_2') \neq (E_1'', E_2'') \Rightarrow (E_1' \cap E_1'' = \emptyset \wedge E_2' \cap E_2'' = \emptyset)$.

Further, we require that all common activities and states are labeled with at least one common label, i.e., for $i \in \{1,2\}$, it holds that $\forall (E_1, E_2) \in \Theta_S \cup \Theta_T, e \in E_i \; \exists (X_1, X_2) \in \Theta_L : \lambda_i(e) \cap X_i \neq \emptyset$, where $\lambda_i : S_i \cup T_i \rightarrow L_i$ is the function describing the labeling of elements in B_i.

$B_i^c = (S_i^c, T_i^c, F_i^c, L_i^c, l_i^c)$, where	$B_i^u = (S_i^u, T_i^u, F_i^u, L_i^u, l_i^u)$, where
$L_i^c = \{x \in L_i \mid \exists (E_1, E_2) \in \Theta_L : x \in E_i\}$	$L_i^u = \{x \in L_i \mid \nexists (E_1, E_2) \in \Theta_L : x \in E_i\}$
$S_i^c = \{s \in S_i \mid \lambda_i(s) \cap L_i^c \neq \emptyset\}$	$S_i^u = \{s \in S_i \mid \lambda_i(s) \cap L_i^u \neq \emptyset\}$
$T_i^c = \{t \in T_i \mid \lambda_i(t) \cap L_i^c \neq \emptyset\}$	$T_i^u = \{t \in T_i \mid \lambda_i(t) \cap L_i^u \neq \emptyset\}$
$F_i^c = F_i \cap ((S_i^c \cup T_i^c) \times (S_i^c \cup T_i^c))$	$F_i^u = F_i \cap ((S_i^u \cup T_i^u) \times (S_i^u \cup T_i^u))$
$l_i^c = \{(f, X) \mid f \in F_i^c \wedge X = l_i(f) \cap L_i^c\}$	$l_i^u = \{(f, X) \mid f \in F_i^u \wedge X = l_i(f) \cap L_i^u\}$

Table 2. Definition of the common and unique sub-views

Example 9. View RES-B is considered as B_1, view RES-R is considered as B_2. There are several equivalences, e.g., $(issue, issue) \in \Theta_T^E$, and inclusions, e.g., the left inclusion $(r, \{rg, rr\}) \in \Theta_L^{I_1}$, and the right inclusion $(\{depToPay, payDeposit, waiveDeposit, depHandled\}, toBill) \in \Theta_S^{I_2}$.

4.5 View Decomposition

In this step, for each enriched view B_i, the *common sub-view* B_i^c and the *unique sub-view* B_i^u are defined: The *common sub-view* consists of all *common labels* as well as activities, states, and arcs that are labeled with at least one common label; the *unique sub-view* consists of all *unique labels* as well as activities, states, and arcs that are labeled with at least one unique label. Formally, B_i^c and B_i^u are defined in Table 2.

Example 10. In the enriched view RES-R (cf. Fig. 2), labels p, rg, and rr and all states, activities, and arcs labeled with at least one of these labels are taken into the common sub-view. The unique sub-view of RES-R consists of label c as well as all states, activities, and arcs that are labeled with c.

4.6 Integration of Common Elements

In this step, the *common view* $\tilde{B}^c$ is defined based on the *common sub-views* B_1^c and B_2^c and the determined correspondences. We will omit definition of the intermediate views $\hat{B}_i^c$ and $\bar{B}_i^c$, but define the common view $\tilde{B}^c = (\tilde{S}^c, \tilde{T}^c, \tilde{F}^c, \tilde{L}^c, \tilde{l}^c)$ immediately from the common sub-views B_1^c and B_2^c. The introduced specialization function h_i^c may be decomposed (cf. Sect. 2.2) to yield the intermediate views.

In $\tilde{B}^c$, a state, an activity, or a label is defined for each common element; we define each element in $\tilde{B}^c$ by an ordered pair that represents the elements of both views that correspond to this element. The refinement function $h_i^c : (\tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c) \rightarrow (S_i^c \cup T_i^c \cup L_i^c)$ is defined for each B_i^c (cf. Tab. 3).

Example 11. The common view of RES-R and RES-B (cf. Figs. 2 and 4 (b)) is defined as shown in Fig. 5, except that the activities and states with dotted borders and the labels put in brackets are not yet included. For better readability, the elements in the common view carry the same name as the corresponding elements in the enriched views RES-B and RES-R.

$\tilde{S}^c := \{(e_1,e_2)\|(e_1,e_2) \in \Theta_S^E\} \cup \{(e_1,e_2)\|\exists(e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in S_2^c \cap E_2\} \cup$ $\{(e_1,e_2)\|\exists(E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in S_1^c \cap E_1\}$
$\tilde{T}^c := \{(e_1,e_2)\|(e_1,e_2) \in \Theta_T^E\} \cup \{(e_1,e_2)\|\exists(e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in T_2^c \cap E_2\} \cup$ $\{(e_1,e_2)\|\exists(E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in T_1^c \cap E_1\}$
$\tilde{L}^c := \{(e_1,e_2)\|(e_1,e_2) \in \Theta_L^E\} \cup \{(e_1,e_2)\|\exists(e_1,E_2) \in \Theta_L^{I_1} : e_2 \in E_2\} \cup$ $\{(e_1,e_2)\|\exists(E_1,e_2) \in \Theta_L^{I_2} : e_1 \in E_1\}$
$h_1^c(e^c) := e_1^c$ and $h_2^c(e^c) := e_2^c$, where $e^c = (e_1^c, e_2^c)$
$\tilde{F}^c := \{(e',e'')\|e' \in \tilde{S}^c \cup \tilde{T}^c \wedge e'' \in \tilde{S}^c \cup \tilde{T}^c \wedge$ $(((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c) \vee$ $((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_2^c(e') = h_2^c(e'')) \vee$ $(h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c))\}$
$\tilde{l}^c := \{((e',e''),X)\|(e',e'') \in \tilde{F}^c \wedge X = \{x \in \tilde{L}^c\|$ $((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'),h_1^c(e'')) \wedge$ $(h_2^c(e'),h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'),h_2^c(e'))) \vee$ $((h_1^c(e'),h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'),h_1^c(e'')) \wedge h_2^c(e') = h_2^c(e'')) \vee$ $(h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'),h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'),h_2^c(e''))))\}\}$

Table 3. Definition of the common view

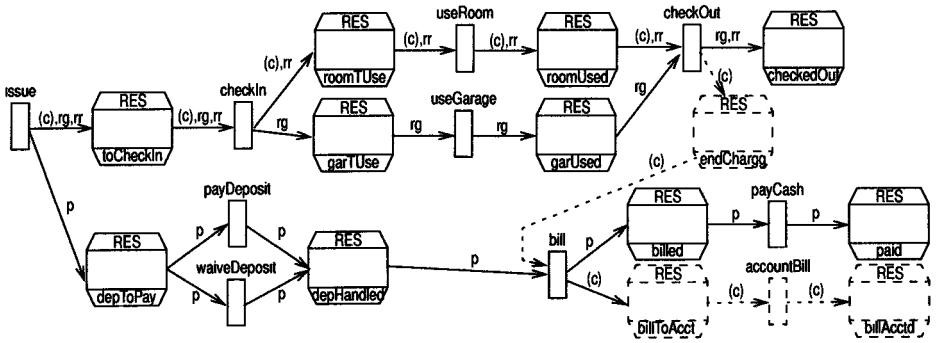


Fig. 5. Integrated business process: Behavior diagram of RES

The common view $\tilde{B}^c$ is correct only if it is an LBD according to Def. 1 and if it is an observation consistent specialization of B_1^c and B_2^c. If these criteria are not fulfilled, the determined correspondences are not correct or the given enriched views are contradictory. An error occurs, for example, if elements that are considered to be corresponding do not behave the same way, e.g., there is an activity and a state in one view that are considered equivalent to an activity and a state, respectively, in the other view; in one view there is an arc between the activity and the state, but in the other view, there is no arc.

Example 12. Consider the case that activity **payCash** has been omitted in RES-B such that only activity **payByMoneyTransfer** is left. This activity does not correspond to **payCash** in view RES-R, but state **paid** in RES-B corresponds to state **paid** in RES-R. Then, the common view would include a state named **paid**, which is not connected to any activity. The reason is that the views cannot be integrated if there is no common means of payment defined.

4.7 Integration of View-Specific Aspects

In the last step, the unique labels are treated to resolve heterogeneities due to *view-specific aspects.*

States and activities that are included in the common sub-view B_i^c as well as the unique sub-view B_i^u of a view B_i and that have been refined in the refined common sub-view $\hat{B}_i^c$ have to be refined in the unique sub-view in the same way (yielding the *refined unique sub-view* $\hat{B}_i^u$). View-specific alternatives that have been omitted in the reduced common sub-view $\bar{B}_i^c$ have to be omitted in the unique sub-view, too (yielding the *reduced unique sub-view* $\bar{B}_i^u$). For all elements of the reduced unique sub-view, a new element is defined in the *unique view* $\tilde{B}_i^u$.

When an activity or a state that is labeled with a unique label is refined in the common view and, thus, in the unique view, the system integrator has to decide how the unique label is treated within the refined state or activity. This problem particularly arises if an activity or state is refined to a subnet including parallel states and activities that enforce splitting of the unique label.

The unique views $\tilde{B}_i^u$ may be defined directly from the unique sub-views B_i^u; $\tilde{B}_i^u$ has to be a correct LBD and an observation consistent specialization of B_i^u based on the specialization function $h_i^u : \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u \rightarrow S_i^u \cup T_i^u \cup L_i^u$.

The specialization of the unique sub-views must not contradict the specialization of the common sub-views. Therefore, the following conditions must hold: (1) $\forall x' \in \tilde{L}_i^u : h_i^u(x') \in L_i^u$ (i.e., labels may be refined but no new labels are introduced), (2) $\forall e \in (S_i^c \cup T_i^c) \cap (S_i^u \cup T_i^u), e' \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{S}_i^u \cup \tilde{T}_i^u : h_i^c(e') = e \Leftrightarrow h_i^u(e') = e$ (i.e., all elements that exist in the common sub-view as well as in the unique sub-view have to be refined in the same way), (3) $\forall e' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u), e'' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u) : (e', e'') \in \tilde{F}^c \Leftrightarrow (e', e'') \in \tilde{F}_i^u$ (i.e., the same set of arcs is defined for elements that exist in the common view as well as in the unique sub-view).

The integrated business process $\tilde{B} = (\tilde{S}, \tilde{T}, \tilde{F}, \tilde{L}, \tilde{l})$ is a composition of $\tilde{B}^c, \tilde{B}_1^u$, and $\tilde{B}_2^u$. It will be an observation consistent specialization of B_1 and B_2, if $\tilde{B}^c$ is an observation-consistent specialization of B_1^c and B_2^c, and for $i \in \{1, 2\}$, $\tilde{B}_i^u$ is an observation-consistent specialization of B_i^u. The specialization function $\tilde{h}_i : \tilde{S} \cup \tilde{T} \cup \tilde{L} \rightarrow S_i \cup T_i \cup L_i \cup \{\varepsilon\}$, which represents the specialization of B_i to $\tilde{B}$ is defined as follows: $\tilde{h}_i(\tilde{e}) := h_i^c(\tilde{e})$ if $\tilde{e} \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c$, $\tilde{h}_i(\tilde{e}) := h_i^u(\tilde{e})$ if $\tilde{e} \in \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u$, and $\tilde{h}_i(\tilde{e}) := \varepsilon$ else.

Example 13. The unique label c of RES-R is taken into the integrated business process RES (cf. Fig. 5).

5 Conclusion

In this paper, we introduced an approach to design a business process for an organization based on views defined by sub-units of the organization. The problem corresponds to the definition of a conceptual database schema based on external views. In this work, we treated the integration of the overall *behavior* of business objects, i.e., the integration of object life-cycles.

Different from the integration of local schemas in federated database systems, where data is shared in the local databases and data in the integrated view is derived, the integration of object life-cycles we have presented assumes that objects are instances of the integrated object type and that views represent derived information on the treatment of the business objects, possibly, restricted to a subset of aspects and at a coarser level of granularity.

We used the criterion of *observation consistent specialization,* which has been introduced for the top-down design of object-oriented databases, to check whether some business process is a correct integration of several views. Different to top-down specialization, integration requires to determine correspondences between elements of views of object life-cycles and to develop an object life-cycle that is a specialization of each given view.

References

1. C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
2. E. Bertino and A. Illarramendi. The Integration of Heterogeneous Data Management Systems: Approaches Based on the Object-Oriented Paradigm. In [4].
3. P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. In *Proc. CAiSE'97*, Springer LNCS 1250, 1997.
4. O. Bukhres and A. Elmagarmid. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications.* Prentice-Hall, 1996.
5. L. Ekenberg and P. Johannesson. A Formal Basis for Dynamic Schema Integration. In *Conceptual Modeling - ER '96*, Springer LNCS 1157, 1996.
6. H. Frank and J. Eder. Integration of Behaviour Models. In *Proc. ER '97 Workshop on Behavioral Models and Design Tranformations*, 1997.
7. G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proc. ICDE'91*, 1991.
8. J. L. Peterson. Petri nets. *ACM Computing Surveys*, pages 223–252, 1977.
9. G. Preuner and M. Schrefl. Observation Consistent Integration of Business Processes. In *Proc. Australian Database Conference (ADC)*, Springer, 1998.
10. M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In *Proc. OO-ER '95*, Springer LNCS 1021, 1995.
11. M. Schrefl and M. Stumptner. Behavior Consistent Refinement of Object Life Cycles. In *Proc. ER '97*, Springer LNCS 1331, 1997.
12. A. Sheth and V. Kashyap. So Far (Schematically) yet So Near (Semantically). In *Proc. DS-5 Semantics of Interoperable Database Systems*, 1992.
13. A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.
14. C. Thieme and A. Siebes. Guiding Schema Integration by Behavioural Information. *Information Systems*, 20(4):305–316, 1995.
15. W. M. P. van der Aalst and T. Basten. Life-Cycle Inheritance — A Petri-Net-Based Approach. In *Proc. PN '97*, Springer LNCS 1248, 1997.
16. M. Vermeer and P. Apers. Behaviour specification in database interoperation. In *Proc. CAiSE '97*, Springer LNCS 1250, 1997.

Efficient Mining of Association Rules in Large Dynamic Databases

Edward Omiecinski and Ashok Savasere

College of Computing, Georgia Institute of Technology, Atlanta GA 30332, USA

Abstract. Mining for association rules between items in a large database of sales transactions is an important database mining problem. However, the algorithms previously reported in the literature apply only to static databases. That is, when more transactions are added, the mining process must start all over again, without taking advantage of the previous execution and results of the mining algorithm. In this paper we present an efficient algorithm for mining association rules within the context of a dynamic database, (i.e., a database where transactions can be added). It is an extension of our *Partition* algorithm which was shown to reduce the I/O overhead significantly as well as to lower the CPU overhead for most cases when compared with the performance of one of the best existing association mining algorithms.

1 Introduction

Increasingly, business organizations are depending on sophisticated decision-making information to maintain their competitiveness in today's demanding and fast changing marketplace. Inferring valuable high-level information based on large volumes of routine business data is becoming critical for making sound business decisions. For example, customer buying patterns and preferences, sales trends, etc, can be learned by analyzing point-of-sales data at supermarkets. Discovering these new nuggets of knowledge is the intent of database mining.

One of the main challenges in database mining is developing fast and efficient algorithms that can handle large volumes of data since most mining algorithms perform computations over the entire database, which is often very large. Besides a large amount of data, another characteristic of most database systems is that they are dynamic (i.e., data can be added and deleted). This poses a potential dilemma for data mining. Since the database will never contain all the data, the data mining process can be performed either on the recent data or on all the data accumulated thus far. In this paper our focus is on applications that need the entire database mined and hence we concentrate on accomplishing this in an incremental fashion.

Discovering association rules between items over *basket* data was introduced in [1]. Basket data typically consists of items bought by a customer along with the date of transaction, quantity, price, etc. Such data may be collected, for example, at supermarket checkout counters. Association rules identify the set of items that are most often purchased with another set of items. For example, an

association rule may state that "95% of customers who bought items A and B also bought C and D." Association rules may be used for catalog design, store layout, product placement, target marketing, etc.

Many algorithms have been discussed in the literature for discovering association rules [1, 2, 3, 4, 5]. One of the key features of all the previous algorithms is that they require multiple passes over the database. For disk resident databases, this requires reading the database completely for each pass resulting in a large number of disk I/Os. In these algorithms, the effort spent in performing just the I/O may be considerable for large databases. For example, a 1 GB database will require roughly 125,000 block reads for a single pass (for a block size of 8KB). If the algorithm takes, say, 10 passes, this results in 1,250,000 block reads. Assuming that we can take advantage of doing sequential I/O where we can read about 400 blocks per second, the time spent in just performing the I/O is approximately 1 hour. If we could reduce the number of passes to two, we reduce the I/O time to approximately 0.2 seconds. With dynamically changing databases, the problem is further compounded by the fact that when new data is added, the mining algorithm must be run again, just as if it were the first time the process was performed. For example, it would require making another 10 passes over the current database.

In a previous paper, we describe our algorithm called *Partition* [6], which is fundamentally different from all the previous algorithms in that it reads the database at most two times to generate all significant association rules. Contrast this with the previous algorithms, where the database is not only scanned multiple times but the number of scans cannot even be determined in advance. Surprisingly, the savings in I/O is not achieved at the cost of increased CPU overhead. We have also performed extensive experiments [6] and compared our algorithm with one of the best previous algorithms. Our experimental study shows that for computationally intensive cases, our algorithm performs better than the previous algorithm in terms of both CPU and I/O overhead.

In this paper we extend our *Partition* algorithm to accomodate dynamically changing databases where new transactions are inserted. We also provide experimental results showing the improved performance over our previous algorithm. We only consider our previous algorithm since it has been shown to have better performance over the best competing algorithm [2] for static databases.

The paper is organized as follows: in the next section, we give a formal description of association mining. In Section 2, we describe the problem and give an overview of the previous algorithms in section 3. In section 4, we describe our algorithm. Performance results are described in section 5. Section 6 contains our conclusions.

2 Association Mining

This section is largely based on the description of the problem in [1] and [2]. Formally, the problem can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of m distinct literals called *items*. $\mathcal{D}$ is a set of variable length transactions

over $\mathcal{I}$. Each transaction *contains* a set of items $i_i, i_j, \ldots, i_k \subset \mathcal{I}$. A transaction also has an associated unique identifier called TID. An *association rule* is an implication of the form $X \Longrightarrow Y$, where $X, Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. X is called the antecedent and Y is called the consequent of the rule.

In general, a set of items (such as the antecedent or the consequent of a rule) is called an *itemset*. The number of items in an itemset is called the *length* of an itemset. Itemsets of some length k are referred to as k–itemsets. For an itemset $X \cdot Y$, if Y is an m–itemset then Y is called an *m-extension* of X.

Each itemset has an associated measure of statistical significance called *support*. For an itemset $X \subset \mathcal{I}$, *support(X)* $= s$, if the fraction of transactions in $\mathcal{D}$ containing X equals s. A rule has a measure of its strength called *confidence* defined as the ratio *support(X* $\cup$ *Y) / support(X)*.

The problem of mining association rules is to generate all rules that have support and confidence greater than or equal to some user specified minimum support and minimum confidence thresholds, respectively. This problem can be decomposed into the following subproblems:

1. All itemsets that have support greater than or equal to the user specified minimum support are generated. These itemset are called the *large* itemsets. All others are said to be *small*.
2. For each large itemset, all the rules that have minimum confidence are generated as follows: for large itemset X and any $Y \subset X$, if *support(X)/support(X* $-Y$*)* $\geq$ *minimum_confidence*, then the rule $X - Y \Longrightarrow Y$ is a valid rule.

For example, let $T_1 = \{A, B, C\}$, $T_2 = \{A, B, D\}$, $T_3 = \{A, D, E\}$ and $T_4 = \{A, B, D\}$ be the only transactions in the database. Let the minimum support and minimum confidence be 0.5 and 0.8 respectively. Then the large itemsets are the following: $\{A\}, \{B\}, \{D\}, \{AB\}, \{AD\}, \{BD\}$ and $\{ABD\}$. The valid rules are $B \Longrightarrow A$ and $D \Longrightarrow A$.

The second subproblem, i.e., generating rules given all large itemsets and their supports, is relatively straightforward. However, discovering all large itemsets and their supports is a nontrivial problem if the cardinality of the set of items, $\mid \mathcal{I} \mid$, and the database, $\mathcal{D}$, are large. For example, if $\mid \mathcal{I} \mid = m$, the number of possible distinct itemsets is 2^m. The problem is to identify which of these large number of itemsets has the minimum support for the given set of transactions. For very small values of m, it is possible to setup 2^m counters, one for each distinct itemset, and count the support for every itemset by scanning the database once. However, for many applications m can be more than 1,000. Clearly, this approach is impractical. To reduce the combinatorial search space, all algorithms exploit the following property: any subset of a large itemset must also be large. Conversely, all extensions of a small itemset are also small. This property is used by all existing algorithms for mining association rules as follows: initially support for all itemsets of length 1 (1-itemsets) are tested by scanning the database. The itemsets that are found to be small are discarded. A set of 2-itemsets called *candidate itemsets* are generated by extending the large 1-itemsets generated in the previous pass by one (1-extensions) and their support is tested by scanning the database. Itemsets that are found to be large are

again extended by one and their support is tested. In general, some kth iteration contains the following steps:

1. The set of candidate k-itemsets is generated by 1-extensions of the large $(k-1)$-itemsets generated in the previous iteration.
2. Supports for the candidate k–itemsets are generated by a pass over the database.
3. Itemsets that do not have the minimum support are discarded and the remaining itemsets are called large k-itemsets.

This process is repeated until no more large itemsets are found.

3 Previous Work Using Static Databases

The problem of generating association rules was first introduced in [1] and an algorithm called *AIS* was proposed for mining all association rules. In [4], an algorithm called *SETM* was proposed to solve this problem using relational operations. In [2], two new algorithms called *Apriori* and *AprioriTid* were proposed. These algorithms achieved significant improvements over the previous algorithms. The rule generation process was also extended to include multiple items in the consequent and an efficient algorithm for generating the rules was also presented.

The algorithms vary mainly in (a) how the candidate itemsets are generated; and (b) how the supports for the candidate itemsets are counted. In [1], the candidate itemsets are generated on the fly during the pass over the database. For every transaction, candidate itemsets are generated by extending the large itemsets from the previous pass with the items in the transaction such that the new itemsets are contained in that transaction. In [2] candidate itemsets are generated in a separate step using only the large itemsets from the previous pass. It is performed by joining the set of large itemsets with itself. The resulting candidate set is further pruned to eliminate any itemset whose subset is not contained in the previous large itemsets. This technique produces a much smaller candidate set than the former technique.

Supports for the candidate itemsets are determined as follows. For each transaction, the set of all candidate itemsets that are contained in that transaction are identified. The counts for these itemsets are then incremented by one. Apriori and AprioriTid differ based on the data structures used for generating the supports for candidate itemsets.

In Apriori, the candidate itemsets are compared with the transactions to determine if they are contained in the transaction. A hashtree structure is used to restrict the set of candidate itemsets compared so that subset testing is optimized. Bitmaps are used in place of transactions to make the testing fast. In AprioriTid, after every pass, an encoding of all the large itemsets contained in a transaction is used in place of the transaction. In the next pass, candidate itemsets are tested for inclusion in a transaction by checking whether the large itemsets used to generate the candidate itemset are contained in the encoding

of the transaction. In Apriori, the subset testing is performed for every transaction in each pass. However, in AprioriTid, if a transaction does not contain any large itemsets in the current pass, that transaction is not considered in subsequent passes. Consequently, in later passes, the size of the encoding can be much smaller than the actual database. A hybrid algorithm is also proposed which uses Apriori for initial passes and switches to AprioriTid for later passes.

In [6] we introduced our Partition algorithm, which reduces the number of database scans to only two. In one scan it generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets, i.e., it may contain false positives. But no false negatives are reported. During the second scan, counters for each of these itemsets are set up and their actual support is measured in one scan of the database.

The algorithm executes in two phases. In the first phase, the Partition algorithm logically divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated. At the end of phase I, these large itemsets are merged to generate a set of all potential large itemsets. In phase II, the actual support for these itemsets are generated and the large itemsets are identified. The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

The results reported in [6] show that the number of page reads can be reduced as much as 87% over the Apriori algorithm [2] and that the overall CPU time can be reduced as much as 81%. The details of the specific experiments are in [6].

4 Partition Algorithm for Dynamic Databases

Since our Partition Algorithm divides the database into any number of partitions for processsing, it seems a perfect starting point for developing an algorithm for mining associations in a dynamically changing database.

Definition A *partition* $p \subseteq \mathcal{D}$ of the database refers to any subset of the transactions contained in the database $\mathcal{D}$. Any two different partitions are non-overlapping, i.e., $p_i \cap p_j = \emptyset, i \neq j$. We define *local support* for an itemset as the fraction of transactions containing that itemset in a partition. We define a *local candidate itemset* to be an itemset that is being tested for minimum support within a given partition. A *local large itemset* is an itemset whose local support in a partition is at least the user defined minimum support (e.g., 2 %, 0.005, etc). A local large itemset may or may not be large in the context of the entire database. We define *global support*, *global large itemset*, and *global candidate itemset* as above except they are in the context of the entire database $\mathcal{D}$. Our goal is to find all global large itemsets.

We can envision two partitions, the original database and the set of transactions to be inserted. Since we assume that the mining algorithm was already run on the original database, we need to take advantage of that by modifying the

Partiton algorithm to create a file containing the global large itemsets along with their counts and the number of records in the current database. This information will allow us to reduce the number of scans on the original database to just one. However, we will have to read the file of global large itemsets instead, which will be much smaller than the original database of transactions. For example, with an original database of 100,000 transactions having an average transaction size of 10 items, we used 4,000,000 bytes of storage. For the same set of data, we computed 93 large itemsets with an average size of 4 items. This produces a large itemset file size of 1488 bytes plus the 4 bytes for each of the 93 large itemset counts and 4 bytes for the total number of records, giving a grand total of 1864 bytes. This is a considerable difference in size and subsequent I/O costs for reading.

Besides modifying our Partition algorithm to save the necessary global large itemset information, we must modify the algorithm to take advantage of this information. As mentioned, the Partition algorithm consists of two phases. In the first phase the local large itemsets are computed along with their counts. Since we are treating the original database as partition 1, we do not have to read that for phase I, but instead read the global large itemset file. Not only does this save I/O time but also CPU time since the large itemsets for that partition do not have to be computed. The next step in phase I is to process the second partition, i.e., the new data. A set of added transactions will be treated differently from a set of deleted transactions. Let's consider the insertion case first, which will be more typical. In the case of insertions, the Partition algorithm works exactly the same as the original. It scans partition 2 (i.e., the inserted transactions) and computes the large itemsets and their counts along with the total number of records to be inserted.

For phase II, we need to see if the local large itemsets of partition 1 and partition 2 will be globally large. This requires reading in the original database once and reading in the set of new transactions once (i.e., for the second time). We could take the union of the local large itemsets for partition 1 and partition 2 and do the processing just as the original algorithm does. However, we can reduce the CPU time by only checking the local large itemsets for partition 1 against the set of new transactions and only checking the local large itemsets for partition 2 against the original database. Of course the number of reads is still the same but the number of comparisons is reduced. In addition, the local large itemsets that are contained in the sets from both partition 1 and partition 2 will in fact be globally large and their counts will simply be the sum of the two local counts. So, we can further reduce the number of comparisons needed in phase II. Hence, counts for the local large itemsets, which are not in the intersection of the two partitions' local large itemsets will be computed during a scan of the approriate data. As with the original algorithm, if the global count for the local large itemsets satisfies the support, then we write that large itemset and its count into our new file of global large itemsets. This file will replace our previous global large itemset file.

We already mentioned that a set of deleted transactions will be processed

differently from a set of inserted transactions. The problem with deletions is that the file of large itemsets may not contain all the large itemsets after deleting the specified records. It is possible that an itemset in the original database was not large but after deleting a certain number of transactions, it becomes large. For example, consider a database of 100 transactions with a minimum support of 10%. It might be that the original count for a given itemset, I, is 9, which is too low to satisfy the support criteria. Now suppose that 10 transactions are deleted and that those 10 transactions are not the ones used in computing the count for itemset I. With a minimum support of 10%, itemset I, would now be considered a large itemset although it did not appear in the original itemset file. Because of this, the original database, excluding the deleted transactions, must be read and processed in phase I, just as in the original Partition algorithm.

However, we may be able to reduce the CPU time for phase I by using the large itemset file. Consider the following: read in the large itemset file and read in the set of deleted transactions. While reading the set of deleted transactions, use this data to compute the counts, for just those large itemsets stored in the large itemset file. We can then subtract these counts from our original counts for the associated large itemsets. If the counts indicate that the support is no longer satisfied, then we can delete that large itemset. In addition, we can use the information that a former large itemset is no longer large for pruning when we have to process the original database minus the deleted transactions to generate large itemsets. The bottom line is whether reading the large itemset file and doing additional processing with the deleted transactions can actually help prune potentially large itemsets from consideration and reduce the CPU costs at this step to produce an overall savings. We leave this for future study.

Algorithm The Dynamic Partition algorithm is shown in Figure 1. Initially the database $\mathcal{D}$ is logically divided into 2 partitions. Actually the set of transactions to be inserted can be further partitoned but for simplicity of presentation, we consider it as only one partition. Phase I of the algorithm takes 2 iterations. During iteration i only partition p_i is considered. The function **gen_large_itemsets** takes partition p_2 as input and generates local large itemsets of all lengths, $L_1^i, L_2^i, \ldots, L_l^i$ as the output. In phase II, the algorithm sets up counters for each global candidate itemset from L^i and counts their support for the partition $p_{i \bmod 2 + 1}$ and generates the global large itemsets.

The key to the correctness of the above algorithm is the same as our original Partition algorithm which is, any potential large itemset appears as a large itemset in at least one of the partitions. A formal proof is given in [7].

4.1 Generation of Local Large Itemsets

The procedure **gen_large_itemsets** takes a partition and generates all large itemsets (of all lengths) for that partition. The procedure is the same as used in our previous work [6] for the insertion case but differs slightly for the deletion case with respect to pruning potential large itemsets. The prune step is performed as follows:

p_1 = the current database of transactions
p_2 = the set of transactions to be inserted
L^i = the set of all local large itemsets in partition i
l = an individual candidate large itemset contained in L_i
L^G_{old} = large itemset file

Phase I
1) read_in_large_itemset_file(L^G_{old}) and store in L^1
2) read_in_partition(p_2)
3) L^2 = gen_large_itemsets(p_2)

Phase II
4) read_in_partition(p_1)
5) for all candidates $l \in L^2$ gen_count(l,p_1)
6) read_in_partition(p_2)
7) for all candidates $l \in L^1$ gen_count(l,p_2)
8) $L^G = \{l \in L^1 \cup L^2 | l.\text{count} \geq minSup\}$
9) write_out_large_itemset_file(L^G_{new})

Fig. 1. Dynamic Partition Algorithm

```
prune (c: k--itemset)
forall (k − 1)--subsets s of c do
     if s ∉ L_{k−1} then
          return ''c can be pruned''
```

The prune step eliminates extensions of $(k-1)$-itemsets which are not found to be large, from being considered for counting support. For example, if L^p_3 is found to be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}, the candidate generation initially generates the itemsets {1 2 3 4} and {1 3 4 5}. However, itemset {1 3 4 5} is pruned since {1 4 5} is not in L^p_3. This technique is the same as the one described in [2] except in our case, as each candidate itemset is generated, its count is determined immediately.

The counts for the candidate itemsets are generated as follows. Associated with every itemset, we define a structure called as *tidlist*. A tidlist for itemset l contains the *TID*s of all transactions that contain the itemset l within a given partition. The *TID*s in a tidlist are kept in sorted order. Clearly, the cardinality of the tidlist of an itemset divided by the total number of transactions in a partition gives the support for that itemset in that partition.

Initially, the tidlists for 1-itemsets are generated directly by reading the partition. The tidlist for a candidate k-itemset is generated by joining the tidlists of the two $(k-1)$-itemsets that were used to generate the candidate k-itemset. For example, in the above case the tidlist for the candidate itemset {1 2 3 4} is generated by joining the tidlists of itemsets {1 2 3} and {1 2 4}.

Correctness It has been shown in [2] that the candidate generation process

correctly produces all potential large candidate itemsets. It is easy to see that the intersection of tidlists gives the correct support for an itemset.

4.2 Generation of Final Large Itemsets

The global candidate set is generated as the union of all local large itemsets from all partitions. In phase II of the algorithm, global large itemsets are determined from the global candidate set. This phase also takes 2 (i.e., the number of partitions) iterations. Initially, a counter is set up for each candidate itemsets and initialized to zero. Next, for each partition, tidlists for all 1-itemsets are generated. The support for a candidate itemset in that partition is generated by intersecting the tidlists of all 1-subsets of that itemset. The cumulative count gives the global support for the itemsets. The procedure **gen_final_counts** is given in Figure 2. Any other technique such as the subset operation described in [2], can also be used to generate global counts in phase II.

```
     C_k^G = set of global candidate itemsets of length k
1)   forall 1-itemsets do
2)      generate the tidlist
3)   for (k = 2; C_k^G ≠ ∅; k++) do begin
4)      forall k--itemset c ∈ C_k^G do begin
5)         templist = c[1].tidlist ∩c[2].tidlist ∩...∩c[k].tidlist
6)         c.count = c.count + | templist |
7)      end
8)   end
```

Fig. 2. Procedure gen_final_counts

Correctness Since the partitions are non-overlapping, a cumulative count over all partitions gives the support for an itemset in the entire database.

4.3 Discovering Rules

Once the large itemsets and their supports are determined, the rules can be discovered in a straight forward manner as follows: if l is a large itemset, then for every subset a of l, the ratio *support(l) / support (a)* is computed. If the ratio is at least equal to the user specified minimum confidence, them the rule $a \Longrightarrow (l-a)$ is output. A more efficient algorithm is described in [2]. As mentioned earlier, generating rules given the large itemsets and their supports is much simpler compared to generating the large itemsets. Hence we have not attempted to improve this step further.

4.4 Size of the Global Candidate Set

The global candidate set contains many itemsets which may not have global support (false candidates). The fraction of false candidates in the global candidate set should be as small as possible otherwise much effort may be wasted in finding the global supports for those itemsets. The number of false candidates depends on many factors such as the characteristics of the data and the size of the original data and the added data which make up the two partitions in our algorithm.

The local large itemsets are generated for the same minimum support as specified by the user. Hence this is equivalent to generating large itemsets with that minimum support for a database which is the same as the partition. So, for sufficiently large partition sizes, the number of local large itemsets is likely to be comparable to the number of large itemsets generated for the entire database. Additionally, if the data characteristics are uniform across partitions, then a large number of the itemsets generated for individual partitions may be common.

Table 1. Comparison of Global Candidate and Final Large Itemsets (support is 0.5 % and 10% data is added)

Itemset Length	# of Global Candidate Large Itemsets	# of Final Large Itemsets
2	929	323
3	1120	297
4	1197	308
5	1075	301
6	759	224
7	393	122
8	139	45
9	30	10
10	1	1
total	5645	1631

The sizes of the local and global candidate sets may be susceptible to data skew. A gradual change in data characteristics, such the average length of transactions, can lead to the generation of a large number of local large sets which may not have global support. For example, due to severe weather conditions, there may be an abnormally high sales of certain items which may not be bought during the rest of the year. If a new partition is made up of data from only this period, then certain itemsets will have high support for that partition, but will be rejected during phase II due to lack of global support. A large number of such spurious local large itemsets can lead to much wasted effort. Another problem is that fewer itemsets will be found common between partitions leading to a larger

Table 2. Comparison of Global Candidate and Final Large Itemsets (support is 0.5 % and 100% data is added)

Itemset Length	# of Global Candidate Large Itemsets	# of Final Large Itemsets
2	361	301
3	452	381
4	538	300
5	398	277
6	228	217
7	122	121
8	45	45
9	10	10
10	1	1
total	2155	1603

global candidate set. These effects can be seen in Table 1 and in Table 2.

In Table 1 we show the variation in the size of the global candidate large itemsets and the final large itemsets produced for differing itemset lengths. The global candidate large itemsets are the result of the union of the local large itemsets from the original data and the added data. The original database contained 13,000 transactions and the added data contained only 1300 transactions. The minimum support was set at 0.05 %. We can compare the results in Table 1 with those shown in Table 2. It can be seen from the two tables that as the size of the added data becomes larger, the variation in the sizes of the global candidate and final large itemsets is reduced dramatically. It should be noted that when the partition sizes of the original and added data are sufficiently large, the local large itemsets and the global candidate itemsets are likely to be very close to the actual large itemsets as it tends to eliminate the effects of local variations in data.

5 Performance Comparison

In this section we describe the experiments and the performance results of our dynamic algorithm and our standard approach. The experiments were run on a Sun Sparc workstation. All the experiments were run on synthetic data. For the experiments, we used the same method for generating the synthetic data sets as in [6].

5.1 Synthetic Data

The synthetic data is said to simulate a customer buying pattern in a retail environment. The length of a transaction is determined by poisson distribution

with mean μ equal to $|T|$. The transaction is repeatedly assigned items from a set of potentially maximal large itemsets, $\mathcal{T}$ until the length of the transaction does not exceed the generated length. The average size of a transaction is 10 items and the average size of a maximal potentially large itemset is 4 items.

The length of an itemset in $\mathcal{T}$ is determined according to poisson distribution with mean μ equal to $|I|$. The items in an itemset are chosen such that a fraction of the items are common to the previous itemset determined by an exponentially distributed random variable with mean equal to a correlation level. The remaining items are randomly picked. Each itemset in $\mathcal{T}$ has an exponentially distributed weight that determines the probability that this itemset will be picked. Not all items from the itemset picked are assigned to the transaction. Items from the itemset are dropped as long as an uniformly generated random number between 0 and 1 is less than a corruption level, c. The corruption level for itemset is determined by a normal distribution with mean 0.5 and variance 0.1.

5.2 Experimental Results

Two different data sets were used for performance comparison. One was a 2 MB file consisting of 50,000 transactions and the other a 1 MB consisting of 25,000 transactions. For both data sets the number of items was set to 1,000.

Figure 3 shows the execution times for the two synthetic datasets. The execution times increase for both the Incremental and Standard algorithms as the minimum support is reduced because the total number of large and candidate itemsets increases. In general, the execution time increases for both the Incremental and Standard algorithms as the minimum support is reduced. The main reason for this is the fact that the total number of candidate and large itemsets increases. In addition, we have already seen that a relatively small size of added data can produce a large number of locally large itemsets during phase I of the algorithm. This increases the cost during phase II when those itemsets are verified against the original database. So, even though the number of final large itemsets is not increased, there is additional time required for testing the false candidate itemsets. As mentioned, this depends heavily on the characteristics of the data.

From the experimental results, we see that a support of 2% and a support of 1% yield approximately the same number of local large candidate itemsets, producing similar execution times, as shown in Figure 4. In the case of 0.5% support, a noticable increase in the number of candidate itemsets and the number of final itemsets was observed, producing a larger execution time. In Table 3 we show the average improvement of the Incremental approach over the Standard approach.

In Figure 5, we show the local large itemsets generated for both the original database and the added data. Since the database is much larger than the added data and closer in size to the final database size after insertions are done, the number of local large itemsets is much closer to the final number of local large

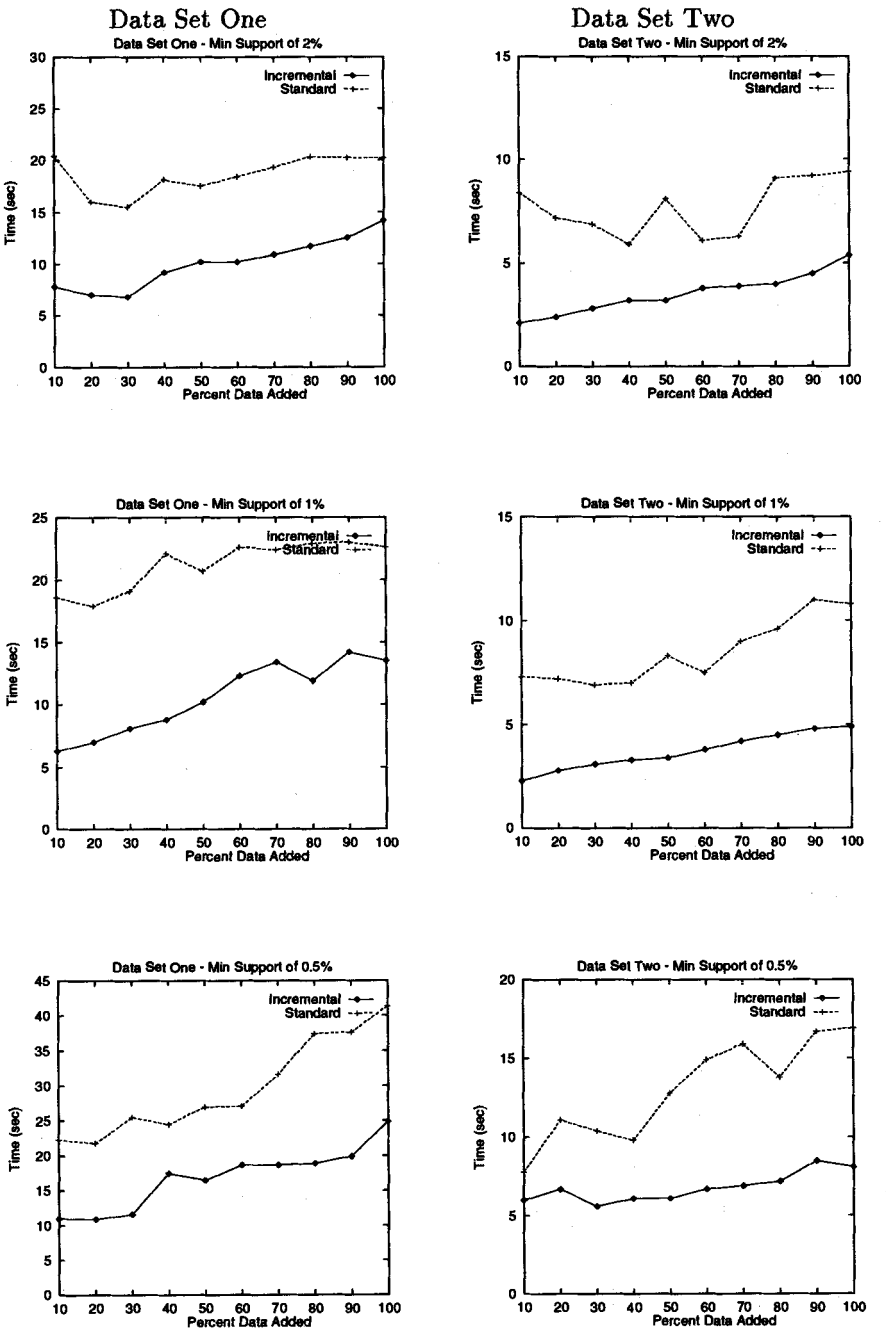


Fig. 3. Execution times

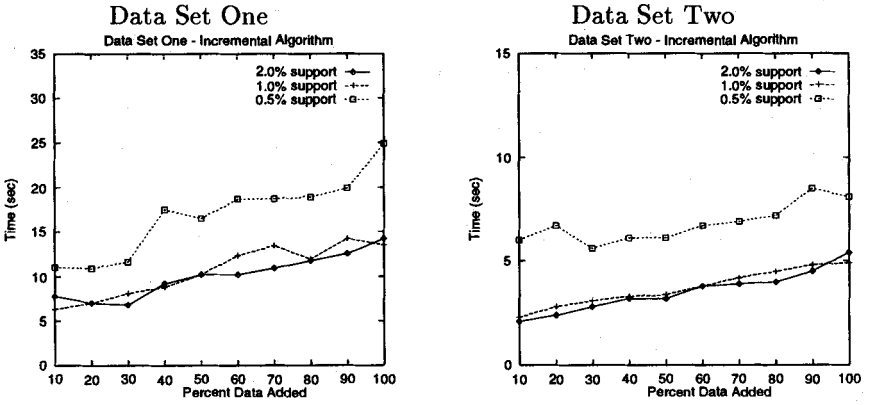


Fig. 4. Execution times for Incremental Algorithm

Table 3. Average Improvement in Execution Time of Incremental Algorithm over Standard Algorithm

Dataset	Minimum Support (%)	Avg. Improvement (%)
1	2.0	46.1
1	1.0	50.7
1	0.5	42.7
2	2.0	52.6
2	1.0	55.0
2	0.5	45.9

itemsets. We see that the number of local large itemsets peaks at 4, which is our average local large itemset size.

6 Conclusions

We have described an algorithm for dynamic data mining, which is an extension to our previous work for mining associations in static databases. We have run several experiments comparing the execution time of our Incremental algorithm and the Standard algorithm. The average improvement (or reduction in execution time) by using our Incremental algorithm ranged from 42.7% to 55%. In our experiments, the database was set to a size such that the Standard algorithm required only one pass over the data. This is the best we can achieve in reducing I/O costs. Where as, our Incremental algorithm required, in phase I, one pass over the added data and one pass over the stored large itemset file, and in phase II, our algorithm required one pass over the original database and one pass over the added data. As we have observed, the major savings from our algorithm

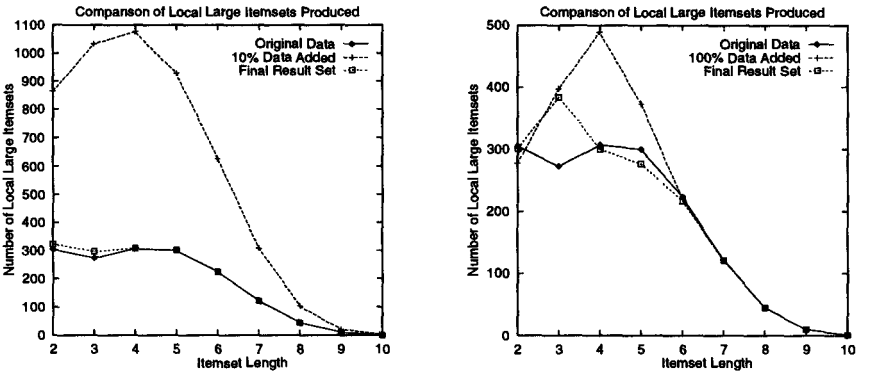


Fig. 5. Large Itemset Comparison

comes about by the fact that the large itemsets for the original database do not have to be re-computed. The substantial savings in execution time makes our Incremental algorithm a desired approach for dynamic data mining.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, May 26-28 1993.
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29-September 1 1994.
3. J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the VLDB Conference*, pages 420 – 431, September 1995.
4. M. Houtsma and A. Swami. Set-oriented mining of association rules. In *Proceedings of the International Conference on Data Engineering*, Taipei, Taiwan, March 1995.
5. J. S. Park, M-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 229 – 248, San Jose, California, May 1995.
6. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 688–192, Zurich, Swizerland, August 1995.
7. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. Technical Report GIT-CC-95-04, Georgia Institute of Technology, Atlanta, GA 30332, January 1995.

Efficient Nearest-Neighbour Searches Using Weighted Euclidean Metrics

Ruth Kurniawati*, Jesse S. Jin, and John A. Shepherd

School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia

Abstract. Building an index tree is a common approach to speed up the k nearest neighbour search in large databases of many-dimensional records. Many applications require varying distance metrics by putting a weight on different dimensions. The main problem with k nearest neighbour searches using weighted euclidean metrics in a high dimensional space is whether the searches can be done efficiently. We present a solution to this problem which uses the bounding rectangle of the nearest-neighbour disk instead of using the disk directly. The algorithm is able to perform nearest-neighbour searches using distance metrics different from the metric used to build the search tree without having to rebuild the tree. It is efficient for weighted euclidean distance and extensible to higher dimensions.

Keywords. nearest-neighbour search, weighted Euclidean distance, coordinate transformations, R-trees, spatial access methods

1 Introduction

The problem of k nearest-neighbour search using a distance metric can be stated as follows: given a collection of vectors and a query vector find k vectors closest to the query vector in the given metric. Nearest-neighbour search is an important operation for many applications. In image databases utilizing feature vectors [13, 12, 7], we want to find k images similar to a given sample image. In instance-based learning techniques [1, 14], we would like to find a collection of k instance vectors most similar to a given instance vector in terms of target attributes. Other applications include time-series databases [6], non-parametric density estimation [9], image segmentation [2], etc.

In the k-nearest-neighbour search process, we have a region defined by the k-th farthest neighbour found so far. Depending on the distance metric used, this area could be a circle (for unweighted euclidean distance), diamond (for manhattan distance), ellipse (for weighted euclidean distance), or other shapes. At the beginning of the search process, this area is initialized to cover the whole

* Corresponding author: E-mail: ruthk@cse.unsw.edu.au; Telephone: 61-2-9385-3989; Fax: 61-2-9385-1813

data space. As we find closer neighbours, the area will get smaller. We call this area "the nearest-neighbour disk"[1]. The search proceeds downward from the top of the tree, checking tree nodes covering smaller area as it deepens. We usually use a branch-and-bound algorithm and try to choose the nodes which are most likely to contain the nearest neighbour first. This is done in order to shrink the nearest-neighbour disk as soon as possible. We can safely ignore nodes whose bounds do not intersect the nearest-neighbour disk.

Many algorithms have been proposed for finding k nearest neighbours more efficiently than a sequential scan through data. These algorithms involve building a spatial access tree, such as an R-tree [10], PMR quadtree [11], k-d-tree [3], SS-tree [22], or their variants [18,8,19,16]. Spatial access methods using such trees generally assume that the distance metric used for building the structures is also the one used for future queries. However, many applications require varying distance metrics. For example, when the the similarity measure is subjective, the distance metric can vary between different users. Another example comes from the instance-based learning [1,14], where the distance metric is learnt by the algorithm. The distance metric will change as more instances are collected. The tree structures above cannot be easily used if the distance metric in the query is different to the distance metric in the index tree. One approach is to map the index tree into the new metric space, but this is equivalent to rebuilding the tree, and therefore, computationally expensive. Another approach is to map the k nearest neighbour bounding disk into the index metric. For example, if we do a query using the Minkowski distance metric on a tree built in euclidean distance, the corresponding nearest neighbour bounding region will be of a diamond shape. Similarly, a nearest neighbour disk in the euclidean distance with different weighting will produce an ellipse. Because of the wide variety of shapes than can be produced, calculating the intersection of the new nearest neighbour bounding envelope and the bounding envelopes of index tree nodes is not trivial and can be computationally expensive.

In this paper, we present an efficient k-nearest-neighbour search algorithm using weighted euclidean distance on an existing spatial access tree. We start by describing a commonly used approach [4]. We identify the drawbacks of this approach and introduce our method. The theoretical foundations and detailed implementation of our method in two dimensional space are given. We also give an extension of the algorithm in higher dimensional spaces. The experiments (up to 32 dimensions) were carried out on an R-tree variant, the SS$^+$-tree [16], (although the proposed approach can be used in any hierarchical structure with a bounding envelope for each node).

[1] We use the 2-dimensional term "disk" regardless of the number of dimensions. To be strict, we should use disk only for 2-dimensional spaces, ellipsoid for 3-dimensional spaces and hyper-ellipsoid for spaces with dimensionality greater than 3.

2 Methods for finding the k nearest-neighbours in weighted euclidean distance

Tree structured spatial access methods organize the data space in a hierarchical fashion. Each tree node in these structures has a bounding hyper-box (R-trees), bounding hyper-planes (k-d-trees), or a bounding hyper-sphere (SS-trees), giving the extent of its child nodes. The root node covers the whole data space which is split among its children. The process is repeated recursively until the set of points in the covered space can fit in one node. To simplify the discussion, we use two dimensional terms for all geometric constructs in the rest of this section and next section.

2.1 Faloutsos' algorithm for k-nn search in a weighted euclidean distance

If the distance metric in the query is consistent with the metric in the index tree, detecting the intersection between the k nearest-neighbour disk and a node can be performed by a simple comparison of the distance of the k-th neighbour to the query vector and the range of the node's bounds in each dimension. When the distance metric is not consistent with the metric in the tree structure, a commonly used technique [4, 5, 15] to find the k-nearest-neighbours is as follows (For easy understanding, we take an example, in which the index tree is built in unweighted euclidean distance and the query is issued in a weighted euclidean distance, and visualize the process in Fig. 1):

- First, we define a distance metric which is a lower bound to the weighted distance. It can be a simple transform to both metrics in the query and in the index tree. Usually, the same metric as in the index tree is used.
- We then search for k nearest neighbours to obtain a bound (LBD in Fig. 1).
- Transferring the distance metric to the query distance metric, we obtain a new bounding shape for the k-nn disk (ED in Fig. 1). The maximum distance D can then be calculated.
- Next, we issue a range query to find all vectors within the distance D from the query point (All vectors in RQD, as shown in Fig. 1). The number of vectors returned from the range query will be greater or equal to k.
- We sort the set from the last step and find the true k nearest neighbours.

From Fig. 1, we can see that the circular range query area (RQD) is much larger than the actual k nearest-neighbour disk. This is not desirable, since a larger query area means we have to examine more tree nodes.

Another drawback of the approach is that we have to search the tree twice. The first scan finds the k nearest-neighbours and the second scan finds all the vectors located within the range D (a range query). Sorting is also needed for selecting first k nearest neighbours within range D.

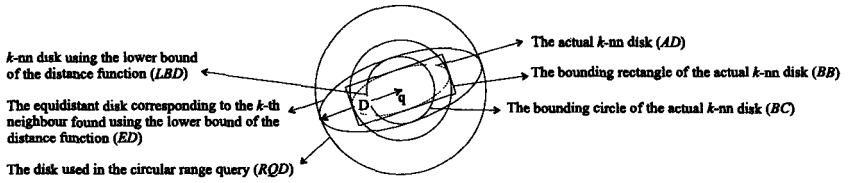


Fig. 1. The evolution of search areas in Faloutsos' approach and our bounding rectangle.

2.2 An efficient k-nn search in variable metrics using the bounding rectangle

We propose an efficient k-nn search with variable metrics. The approach is based on the observation that a transform between the query metric and the index metric exists and transforming the k-nn disk is much simpler than transforming all tree nodes. Using this transformation, we can find the length of the major axis of the k nearest-neighbour disk[2]. However, conducting the search using the ellipse directly is not always desirable. For d-dimensional vectors we will have to calculate the intersection of $(d-1)$-dimensional hyper-planes and a d-dimensional hyper-ellipse. If the intersection is not empty, the result will be a $(d-1)$-dimensional hyper-ellipse. Hence in our solution we use a simpler shape to "envelope" the k nearest-neighbour disk.

We propose a rectangular envelope (the bounding rectangle BB of the k nearest-neighbour disk is shown in Fig. 1). The bounding rectangle can be obtained from calculating the length of all axes of the k-nearest-neighbour disk. Another simpler alternative is by calculating the bounding circle of the k nearest-neighbour disk (BC in Fig. 1). The radius of the bounding circle is equal to the half of ellipse's major axis. Theoretical analysis and detailed calculation will be described in Sect. 3.

Comparing with Faloutsos' algorithm, our approach has two advantages. The bounding rectangle is much compact than Faloutsos' query disk (see RQD and BB/BC in Fig. 1). The advantage is significant if the ratio between the largest and the smallest axes of the nearest neighbour disk is large. There is no need for the second search which is also a significant saving if the level of the tree is high.

3 Theory and implementation

In this paper, we will limit our discussion to the *weighted euclidean distance* $\mathrm{d}(\mathbf{x},\mathbf{y},\mathbf{W}) = \sqrt{((\mathbf{x}-\mathbf{y})^T\mathbf{W}(\mathbf{x}-\mathbf{y}))}$ with a symmetrical weight matrix $\mathbf{W}$. The approach described in this paper can be adapted directly to distances in the

[2] Some books define the axis length of an ellipse as the distance from the centre to perimeter along the axis. The definition we use is the distance from perimeter to perimeter along that axis.

class $\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{i=1}^{d}(x_i - y_i)^p$, where d is the number of the dimensions and $p = 1, 2, 3, \ldots$. We require that the matrix $\mathbf{W}$ is positive definite, i.e. it has positive pivots, which implies that we will always be able to decompose the matrix $\mathbf{W}$ into $\mathbf{Q}^T\mathbf{Q}$ using Cholesky factorization [20]. In practice, we usually deal with symmetrical weight matrices, since non-symmetrical weight matrices means that the distance measured from $\mathbf{x}$ to $\mathbf{y}$ is different from that measured from $\mathbf{y}$ to $\mathbf{x}$.

As have been defined informally in Sect. 2, the *nearest-neighbour disk* is the area bounded by the equidistant points with the current k-th nearest point to the query measured using a particular distance metric. The term "point" used here and in the rest of the paper is interchangeable with "vector".

A particular distance metric will define a *space* where the distance is used for measurements. An euclidean distance defines an euclidean space. The k nearest-neighbour disk is circular in the same euclidean space. If the Euclidean metric in query has a different weight on the metric used in the index, the k nearest-neighbour disk is an ellipse in that space.

3.1 Theoretical foundations

Assume we have two points $\mathbf{x_1}$ and $\mathbf{y_1}$ in a space S_1 where we use unweighted euclidean distance (see Fig. 2). Applying a transformation defined by the matrix $\mathbf{Q}$ will bring every point in S_1 into another space S_2. Suppose $\mathbf{x_1}$ and $\mathbf{y_1}$ are mapped into $\mathbf{x_2}$ and $\mathbf{y_2}$, respectively. If the transformation includes scaling and rotation operations, then the distance between $\mathbf{x_1}$ and $\mathbf{y_1}$ in S_1 will equal a weighted euclidean distance between $\mathbf{x_2}$ and $\mathbf{y_2}$ where the weight matrix $\mathbf{W}$ equals $\mathbf{Q}^T\mathbf{Q}$. This means that a circle in S_1 will be transformed into an ellipse in S_2 since the equivalent distance in S_2 is a weighted euclidean distance.

Observation 1. *The weighted distance* $d(\mathbf{x_2}, \mathbf{y_2}, \mathbf{W})$ *with* $\mathbf{W} = \mathbf{Q}^T\mathbf{Q}$ *equals* $d(\mathbf{x_1}, \mathbf{y_1}, \mathbf{I})$, *where* $\mathbf{x_2} = \mathbf{Q}\mathbf{x_1}$, $\mathbf{y_2} = \mathbf{Q}\mathbf{y_1}$, *and* $\mathbf{I}$ *is the identity matrix.*

Proof. Let $\mathbf{W} = \mathbf{Q}^T\mathbf{Q}$, $\mathbf{x_1} = \mathbf{Q}\mathbf{x_2}$, and $\mathbf{y_1} = \mathbf{Q}\mathbf{y_2}$.

$$\begin{aligned}
\mathrm{d}(\mathbf{x_2}, \mathbf{y_2}, \mathbf{W}) &= \sqrt{(\mathbf{x_2} - \mathbf{y_2})^T\mathbf{W}(\mathbf{x_2} - \mathbf{y_2})} \\
&= \sqrt{(\mathbf{x_2} - \mathbf{y_2})^T\mathbf{Q}^T\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2})} = \sqrt{(\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2}))^T\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2})} \\
&= \sqrt{(\mathbf{Q}\mathbf{x_2} - \mathbf{Q}\mathbf{y_2})^T(\mathbf{Q}\mathbf{x_2} - \mathbf{Q}\mathbf{y_2})} = \sqrt{(\mathbf{x_1} - \mathbf{y_1})^T(\mathbf{x_1} - \mathbf{y_1})} \\
&= \mathrm{d}(\mathbf{x_1}, \mathbf{y_1}, \mathbf{I})
\end{aligned}$$

The transformation matrix $\mathbf{Q}$ defines the scaling and rotation operations which map the circular k nearest-neighbour disk in unweighted euclidean space into an ellipse in the weighted euclidean space. This ellipse is defined by all axes and the query point. As described in Sect. 2, to obtain the bounding box, we need to calculate the length of all the axes and the orientation of the major axis. The solution can be derived from Theorem 2.

Theorem 2. *If we apply a transformation* $\mathbf{Q}$ *to all points of a circle (with radius* r*), the resulting points form an ellipse whose centre is the same as the circle and the length of its axes equals* $2r$ *times the square root of eigenvalues of* $\mathbf{Q}^T\mathbf{Q}$.

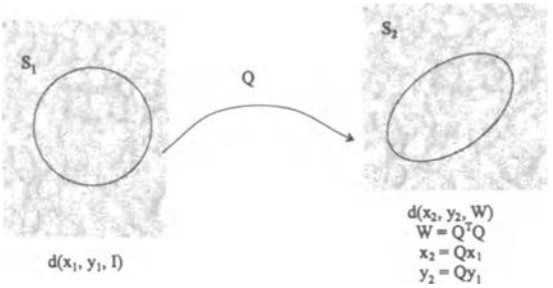


Fig. 2. The transformation between a normal euclidean space and a weighted euclidean space.

Proof. We provide the proof for the two dimensional case. A general proof can be found in [17].

The proof of this theorem depend on a lemma stating that *a real symmetric $n \times n$ matrix has n real eigenvalues, and n linearly independent and orthogonal eigenvectors.* The proof of the lemma can be found in [21].

Let $\mathbf{k_2} = \mathbf{Qk_1}$ where $\mathbf{k_1}$ is an arbitrary point in our original space (S_1 as shown in Fig. 2). Let the circle centre $\mathbf{q_2} = \mathbf{Qq_1}$. The squared distance of $\mathbf{k_2}$ and $\mathbf{q_2}$ is $(\mathbf{Qk_1} - \mathbf{Qq_1})^T \cdot (\mathbf{Qk_1} - \mathbf{Qq_1}) = (\mathbf{k_1} - \mathbf{q_1})^T\mathbf{Q}^T\mathbf{Q}(\mathbf{k_1} - \mathbf{q_1})$. Since $\mathbf{Q}^T\mathbf{Q}$ is real and symmetric, according to the lemma, it has two real eigenvalues and two orthogonal eigenvectors. Let $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ be the unit eigenvectors corresponding to eigenvalues λ_1 and λ_2, respectively.

These orthogonal eigenvectors span $\Re^2$, and hence $(\mathbf{k_1} - \mathbf{q_1})$ can be written as a linear combination of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. Suppose $(\mathbf{k_1} - \mathbf{q_1}) = r\cos(\theta)\boldsymbol{e}_1 + r\sin(\theta)\boldsymbol{e}_2$.

$$
\begin{aligned}
&(\mathbf{k_2} - \mathbf{q_2})^T(\mathbf{k_2} - \mathbf{q_2}) = (\mathbf{k_1} - \mathbf{q_1})^T\mathbf{Q}^T\mathbf{Q}(\mathbf{k_1} - \mathbf{q_1}) \\
&= (r\cos\theta\boldsymbol{e}_1^T + r\sin\theta\boldsymbol{e}_2^T)(r\cos\theta\mathbf{Q}^T\mathbf{Q}\boldsymbol{e}_1 + r\sin\theta\mathbf{Q}^T\mathbf{Q}\boldsymbol{e}_2) \\
&= (r\cos\theta\boldsymbol{e}_1^T + r\sin\theta\boldsymbol{e}_2^T)(r\cos\theta\lambda_1\boldsymbol{e}_1 + r\sin\theta\lambda_2\boldsymbol{e}_2) \\
&= r^2\lambda_1\cos^2\theta + r^2\lambda_2\sin^2\theta
\end{aligned}
$$

Square distance $r^2\lambda_1\cos^2\theta + r^2\lambda_2\sin^2\theta$ has its extreme values at $\theta =$ multiplies of $\frac{\pi}{2}$ – that is, when we are in $\boldsymbol{e}_1$ or $\boldsymbol{e}_2$ directions. The values at those points are $r^2\lambda_1$ and $r^2\lambda_2$, respectively. Using our definition of the ellipse axis, the length of two axes are $2r\sqrt{\lambda_1}$ and $2r\sqrt{\lambda_2}$, respectively.

□

We can write the weight matrix in diagonal form: $\mathbf{W} = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{-1}$, where $\mathbf{E}$ is an orthonormal matrix whose columns are the eigenvectors of $\mathbf{W}$ and $\boldsymbol{\Lambda}$ is a diagonal matrix of the corresponding eigenvalues. Since $\mathbf{E}$ is orthonormal, then $\mathbf{E}^{-1} = \mathbf{E}^T$. Hence:

$$
\begin{aligned}
\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W}) &= \sqrt{((\mathbf{x} - \mathbf{y})^T\mathbf{W}(\mathbf{x} - \mathbf{y}))} \\
&= \sqrt{((\mathbf{x} - \mathbf{y})^T\mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{-1}(\mathbf{x} - \mathbf{y}))} \\
&= \sqrt{((\mathbf{E}^{-1}\mathbf{x} - \mathbf{E}^{-1}\mathbf{y})^T\boldsymbol{\Lambda}(\mathbf{E}^{-1}\mathbf{x} - \mathbf{E}^{-1}\mathbf{y}))}
\end{aligned}
$$

We can see the last line of the equation as another weighted distance with diagonal weight matrix $\mathbf{\Lambda}$. Because $\mathbf{\Lambda}$ is diagonal, the nearest neighbour disk in this space has axes parallel to the coordinate axes. The size of the nearest neighbour disk is the same in both spaces since $\mathbf{E}$ is orthonormal. Suppose $\mathbf{x}_\mathrm{l}$ and $\mathbf{y}_\mathrm{l}$ are in this space, then $\mathbf{x}_\mathrm{l} = \mathbf{E}^{-1}\mathbf{x}$ and $\mathbf{y}_\mathrm{l} = \mathbf{E}^{-1}\mathbf{y}$. We can see $\mathbf{E}$ as the rotation matrix that rotates the nearest neighbour disk. Since $\mathbf{x} = \mathbf{E}\mathbf{x}_\mathrm{l}$ and $\mathbf{y} = \mathbf{E}\mathbf{y}_\mathrm{l}$, then the desired rotation is $\mathbf{E}$. The length of the disk's axis in the direction given by the i-th column of $\mathbf{E}^{-1}$ is given by $\sqrt{(\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W})/\lambda_i)}$.

3.2 Calculating the bounding circle of the k nearest-neighbour disk

In a k nearest-neighbour search using weighted euclidean distance, we start from an initial disk covering the whole data space. After we find k neighbours from searched nodes, a new (smaller) k-nn disk is obtained. We have to calculate the bounding circle of the k-nn disk every time we find a neighbour nearer to the query point than the k-th nearest neighbour found so far. The problem can be stated as follows: given the query point $\mathbf{q}$ and the newly found k-th neighbour we need to calculate the radius of the bounding circle of the ellipse with $\mathbf{q}$ as its centre and $\mathbf{k}$ on its perimeter (see Fig. 3).

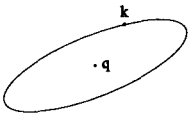


Fig. 3. The query point $\mathbf{q}$ and its k-th nearest-neighbour $\mathbf{k}$.

Based on the discussion in Sect. 3.1, the calculation of the bounding circle of the k nearest-neighbour disk is done as follows.

- Let λ be the largest eigenvalue of matrix $\mathbf{W}$.
- The bounding circle is a circle with the query point $\mathbf{q}$ as its centre and r as its radius, where $r = \mathrm{d}(\mathbf{q}, \mathbf{k}, \mathbf{W})/\sqrt{\lambda_{\min}}$, where $\lambda_{\min}$ is the smallest eigenvalue.

3.3 Calculating the bounding rectangle of the k nearest-neighbour disk

Instead of using bounding circle, we also can use bounding rectangle. The problem, similar to the one described in Sect. 3.2 can be stated as follows: given a query point $\mathbf{q}$ and the newly found k-th neighbour $\mathbf{k}$, calculate the length, the sides and the orientation of the bounding rectangle of the ellipse centred at $\mathbf{q}$ with $\mathbf{k}$ on its perimeter (see Fig. 3).

Let $\mathbf{W}$ is the weight matrix, and $\mathbf{I}$ is the identity matrix, $\mathbf{q}$ is the query point, $\mathbf{k}$ is the k-th nearest neighbour found, $\mathbf{E}$ is an orthonormal matrix whose columns is the eigenvectors of $\mathbf{W}$, λ_i is the eigenvalues corresponding to the

eigenvector in column i of $\mathbf{E}$, and $\boldsymbol{e}_i$ is the vector in column i of $\mathbf{E}^{-1}$. The bounding rectangle of the nearest-neighbour disk can be calculated as follows.

- Let $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ be the eigen-vectors of $\mathbf{W}$ and let λ_1 and λ_2 be the corresponding eigen-values sorted according to their magnitudes.
- Then the bounding rectangle R is defined by the centre $\mathbf{q}$ and dimensional vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ where $r = \mathrm{d}(\mathbf{q}, \mathbf{k}, \mathbf{W})$, and $\boldsymbol{v}_1 = \boldsymbol{e}_1 r/\sqrt{\lambda_1}$ and $\boldsymbol{v}_2 = \boldsymbol{e}_2 r/\sqrt{\lambda_2}$. The corners of the rectangles are:

$$\begin{aligned}
\mathbf{r}_1 &= \mathbf{q} + \boldsymbol{v}_1 + \boldsymbol{v}_2 \\
\mathbf{r}_2 &= \mathbf{q} + \boldsymbol{v}_1 - \boldsymbol{v}_2 \\
\mathbf{r}_3 &= \mathbf{q} - \boldsymbol{v}_1 - \boldsymbol{v}_2 \\
\mathbf{r}_4 &= \mathbf{q} - \boldsymbol{v}_1 + \boldsymbol{v}_2
\end{aligned}$$

Vectors $\boldsymbol{v}_i$ are in the directions parallel to the axes of the nearest neighbour disk. The fact that $\boldsymbol{v}_i$ is equal to $\boldsymbol{e}_i r/\sqrt{\lambda_i}$ can is explained in Sect. 3.1. Note that the rectangle R can be defined by the centre point $\mathbf{q}$ and the directional vectors denoted as $R(\mathbf{q}, \boldsymbol{v}_1, \boldsymbol{v}_2)$ and does not necessarily have any side parallel to the coordinate axes.

In high dimensional space, care must be taken in calculating the intersection between the bounding hyper-box and tree's bounding envelopes. A naïve calculation is exponential to the dimension. We provide an algorithm with time complexity $\mathrm{O}(d^3)$ of dimension d in Sect. 4.

4 Extension to a higher dimensional space

The algorithm in Sect. 3.3 can be used directly in space with dimensionality greater than 2. It contains three major computations, calculating Cholesky decomposition, finding the bounding hyper-box of the new k-nn hyper-ellipsoid, and checking the intersection of a node and the hyper-box. In this section we describe an efficient extension which gives $\mathrm{O}(d^3)$, $\mathrm{O}(d^2)$ and $\mathrm{O}(d^3)$ complexities to three steps respectively.

Note that checking if a node is intersected with the bounding hyper-box does not need to calculate the corner points of the bounding hyper-box. Otherwise, the approach will not be able to scale up to high dimensions because the number of corners of a hyper-box in a d-dimensional space is 2^d. We can define a bounding hyper-box using a centre point and d directional vectors $(\boldsymbol{r}_i)$. Since the Cholesky decomposition and the eigen matrix are the same throughout the search process, we only need to keep the radius r. The centre of the hyper-box will be the query point $\mathbf{q}$ itself. The cost to calculate the eigen matrix and Cholesky decomposition of a d x d matrix is $\mathrm{O}(d^3)$. We only need to recalculate these matrices whenever the weight matrix changes, which can only happen between queries. The calculation of the bounding rectangle for the query area can be done in $\mathrm{O}(d^2)$.

Our algorithm to check the intersection of tree nodes and the bounding hyper-box of the nearest-neighbour disk has a complexity $\mathrm{O}(d^3)$. If the bounding

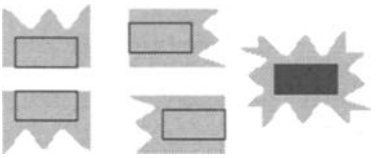


Fig. 4. A bounding rectangle is the intersection of four half-spaces.

hyper-box has sides parallel/perpendicular to the coordinate axes, intersections can be calculated in $O(d)$. We only need to calculate the intersection of each dimension. It can be done by utilizing the fact that a d-dimensional hyper-box is actually an intersection of $2d$ hyper-planes (as shown in the 2D case in Fig. 4).

Suppose the query's bounding hyper-box B is defined by its centre point $\mathbf{q}$ and d orthogonal vectors $\boldsymbol{v}_i$, ie, $B(\mathbf{q}, \boldsymbol{v}_1, ..., \boldsymbol{v}_d)$. The bounding hyper-box of a tree node is A and its closest corner to the origin is $\mathbf{a_{min}}$ and the farthest corner to the origin is $\mathbf{a_{max}}$. We use $\mathbf{r}(j)$ and $\boldsymbol{r}(j)$ to represent the projections of point $\mathbf{r}$ and vector $\boldsymbol{r}$ in dimension j respectively in the following description.

- For all dimensions $j = 1..d$, find the minimum coordinate of the query's bounding hyper-box (can be done in $O(d)$ by subtracting $\boldsymbol{v}_i(j)$ from $\mathbf{q}(j)$). If any minimum is not inside the half-space defined by the equation $x \leq \mathbf{a_{max}}(j)$, then we do not have an intersection. Otherwise, continue checking. This process takes $O(d^2)$ comparisons.
- For all dimension $j = 1..d$, find the maximum coordinate of the query's bounding hyper-box (can be done in $O(d)$ by adding $\boldsymbol{v}_i(j)$ to $\mathbf{q}(j)$). If any minimum is not inside the half-space defined by the equation $x \geq \mathbf{a_{min}}(j)$, then we do not have an intersection. Otherwise, continue checking. This process also takes $O(d^2)$ comparisons.
- If above checking fails, it does not mean that we have a non-empty intersection area. It means that B is not intersected with the hyper-box $\mathbf{a_{min}} \leq x \leq \mathbf{a_{max}}$. The check should also be made from the B rectangle in the query space using all the half-spaces defined by $x \leq \mathbf{q}+\boldsymbol{v}_i$ and $x \geq \mathbf{q}-\boldsymbol{v}_i$. Transformation to the query space takes $O(d^3)$ multiplication and rest comparisons are $O(d^2)$. The total complexity is $O(d^3) + O(d^2) = O(d^3)$.

5 Empirical tests and discussion

We implemented the algorithm described in Sect. 3.3and 3.2 on top of an R-tree variant, the SS$^+$-tree [16]. We compare our algorithms with Faloutsos' method using the unweighted euclidean distance as the lower bound of the weighted distance. We evaluate the performance of the methods using the number of nodes touched/used. For all the experiments in this paper we choose to expand the node whose centroid is closest to the query point with respect to the currently used distance metric. Based on our experience [16], this criterion truncates branches

more efficiently than other criteria, e.g. the minimum distance and the minimum of the maximum distance [18].

All the experiments in Table 1 are in 30 dimensions using 14,016 texture feature vectors extracted from images in Photo Disc's image library using the Gabor filter [12]. The experiments was done using a weight matrices whose ratio between the largest and smallest eigenvalues range from 2 to 32. The fan out of tree's leaf nodes is 66 and the fan out of tree's internal nodes is 22.

We measured the number of leaf nodes that has to be examined (*ltouched*), the number of internal nodes accessed (*inode*), and the number of leaf nodes that actually contain any of the nearest-neighbour set (*lused*). The ratio column is the ratio between square root of the largest and the smallest eigenvalues. All the experiments are 21-nearest-neighbour searches and the tabulated results are the average of 100 trials using a random vector chosen from the dataset.

Table 1. A comparison between the transformation and lower-bound method for the texture vectors of Photo Disc's images (14016 vectors, 30 dimensions, node size: 8192 bytes).

	Bounding (hyper)box			Bounding (hyper)sphere			Faloutsos' method		
ratio	ltouch	lused	inode	ltouch	lused	inode	ltouch	lused	inode
2	25.32	7.06	4.22	19.08	7.32	4.20	36.62	13.78	8.16
4	23.38	6.43	4.17	17.36	7.02	4.24	32.54	12.67	8.09
8	26.04	6.45	4.45	18.04	7.00	4.11	35.63	12.66	8.51
16	24.53	6.76	4.08	17.76	7.16	4.24	33.92	13.43	7.97
32	22.89	6.49	4.15	18.48	7.15	4.12	30.08	12.59	8.01

As can be seen in Table 1, our method outperformed Faloutsos' method in all the ratio values we tried. The number of leaf nodes used (*lused*) by our method is 50% of the number used by Faloutsos' method because our method only need to check the tree once. The same explanation applies to the number of internal nodes accessed (*inode*). The number of leaf nodes that have to be accessed by our method is consistently below 75% of the total leaf nodes accessed by the other method. The data suggest that the ratio between the largest and the smallest eigenvectors does not have any effect in the number of nodes accessed for the texture vectors.

The result experimental results using bounding spheres are somewhat counter intuitive. From our intuition, the results should be worse than the ones using bounding boxes. In two and three dimensions, the bounding rectangle/box of an disk occupy a smaller area/volume than the disk's bounding circle/sphere. Our analytical results [17] supports this empirical results. It shows that in high-dimensional space the query sensitive area resulting from using bounding hyper-spheres is smaller than the one resulting from using bounding hyper-boxes.

The experiments in Table 2 are performed using 100,000 uniformly distributed data in 2, 4, 8, 16, and 32 dimensions. The node size was kept fixed

at 8192 bytes, hence the leaf fan-out (*LFO*) and internal node fan-out (*IFO*) decreases as the dimension increases. The ratio between the largest and smallest eigenvalues was fixed at 4 for all the experiments.

Table 2. A comparison between the transformation and lower-bound method for 100,000 uniformly distributed vectors (ratio used: 4, bucket size: 8192 bytes).

			Bounding (hyper)box			Bounding (hyper)sphere			Faloutsos' method		
dimension	LFO	IFO	ltouch	lused	inode	ltouch	lused	inode	ltouch	lused	inode
2	682	255	1.4	1.4	1	1.4	1.4	1	3.2	2.7	2.0
4	409	146	4.5	2.7	2.2	3.6	2.4	2.3	12.5	5.5	4.5
8	227	78	89.5	8.9	4.8	39.5	7.0	4.0	175.1	17.1	10.4
16	120	40	740.6	25.9	20.4	712.5	25.9	22.0	1489.2	46.0	43.8
32	62	20	1613.0	41.0	87.0	1613	42.0	87.0	3226.0	88.0	174.0

In Table 2, the number of nodes accessed are much higher in high dimensions. This is partly due to the smaller internal/leaf node fan-outs (we kept the node size fixed for all dimensions). Similar to the experimental results using texture vectors, the number of leaf nodes and internal nodes used by our method (*lused* and *inodes*) is approximately half of the number used by Faloutsos' method. In terms of the number of leaf nodes accessed (*ltouch*), our method consistently accessed less than 50% of the number accessed by Faloutsos' method when the dimension is greater than four.

The result experimental results using bounding spheres (up to dimension 16) further support the fact that the query sensitive area resulting from using bounding hyper-spheres will get smaller than the one resulting from using bounding hyper-boxes. But the experimental results in 32 dimension shows that the performance is almost the same for both method. This is due to the inherent problem with hierarchical tree access method in high dimension. In this case, the search actually has degraded into linear search (the variations are due to the randomization used in the experiments).

The experiments are verified by comparing the results of the k nearest neighbour search with the results obtained via simple linear search and comparing the results of the two methods.

6 Conclusion

Nearest-neighbour search on multi-dimensional data is an important operation in many areas including multimedia databases. The similarity measurement usually involves weighting on some of the dimensions and weighting may vary between queries. Existing tree structured spatial access methods cannot support this requirement directly. We have developed an algorithm to support variable distance metrics in queries. The algorithm uses the bounding hyper-box of the k-nn hypersphere instead of calculating intersections directly. It has $O(d^3)$ complexity. We

derive the theoretical foundation of the algorithm and give a detailed implementation in the 2D case. We also extend the algorithm to higher dimensions and give a heuristic algorithm to detect the intersection between the query disk and the bounding envelopes of the tree nodes. We provide analytical and empirical results regarding the performance of our approach in terms of the number of disk pages touched. The experiments go up to 32 dimensions. The algorithm has a significant impact on k-nn search and various index trees. There are also research issues in speeding up the transformation. As we can see from the derivation of the algorithm, the major computational cost is in cross-space transformation.

References

1. David W. Aha. A study of instance-based algorithms for supervised learning tasks: Mathematical, empirical, and psychological evaluations (dissertation). Technical Report ICS-TR-90-42, University of California, Irvine, Department of Information and Computer Science, November 1990.
2. S. Belkasim, M. Shridhar, and M. Ahmadi. Pattern classification using an efficient KNNR. *Pattern Recognition*, 25(10):1269–1274, 1992.
3. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.
4. Christos Faloutsos. *Searching Multimedia Databases by Content.* Advances in Database Systems. Kluwer Academic Publishers, Boston, August 1996.
5. Christos Faloutsos, William Equitz, Myron Flickner, Wayne Niblack, Dragutin Petkovic, and Ron Barber. Efficient and effective querying by image content. *J. of Intelligent Information Systems*, 3:231–262, July 1994.
6. Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–429, May 1994.
7. Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Bryan Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, pages 23–32, September 1995.
8. Jerome H. Friedman, Jon Louis Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. on Math. Software (TOMS)*, 3(3):209–226, September 1977.
9. Keinosuke Fukunaga and Larry D. Hostetler. Optimization of k-nearest-neighbor density estimates. *IEEE Transactions on Information Theory*, IT-19(3):316–326, May 1973.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.
11. Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In Max J. Egenhofer and John R. Herring, editors, *Advances in Spatial Databases, 4th International Symposium, SSD'95*, volume 951 of *Lecture Notes in Computer Science*, pages 83–95, Berlin, 1995. Springer-Verlag.
12. Jesse Jin, Lai Sin Tiu, and Sai Wah Stephen Tam. Partial image retrieval in multimedia databases. In *Proceedings of Image and Vision Computing New Zealand*, pages 179–184, Christchurch, 1995. Industrial Research Ltd.

13. Jesse S. Jin, Guangyu Xu, and Ruth Kurniawati. A scheme for intelligent image retrieval in multimedia databases. *Journal of Visual Communication and Image Representation*, 7(4):369–377, 1996.
14. D. Kibler, D. W. Aha, and M. Albert. Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5:51–57, 1989.
15. Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, and Eliot Siegel. Fast nearest-neighbor search in medical image databases. In *International Conference on Very Large Data Bases*, Bombay, India, Sep 1996.
16. Ruth Kurniawati, Jesse S. Jin, and John A. Shepherd. The SS$^+$-tree: An improved index structure for similarity searches in a high-dimensional feature space. In *Proceedings of the SPIE: Storage and Retrieval for Image and Video Databases V*, volume 3022, pages 110–120, San Jose, CA, February 1997.
17. Ruth Kurniawati, Jesse S. Jin, and John A. Shepherd. Efficient nearest-neighbour searches using weighted euclidean metrics. Technical report, Information Engineering Department, School of Computer Science and Engineering, University of New South Wales, Sydney 2052, January 1998.
18. Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, California, May 1995.
19. Robert F. Sproull. Refinements to nearest-neighbour searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991.
20. Gilbert Strang. *Introduction to applied mathematics.* Wellesley-Cambridge Press, Wellesley, MA, 1986.
21. Gilbert Strang. *Linear algebra and its applications.* Harcourt, Brace, Jovanovich, Publishers, San Diego, 1988.
22. David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proc. 12th IEEE International Conference on Data Engineering*, New Orleans, Louisiana, February 1996.

The Acoi Algebra: A Query Algebra for Image Retrieval Systems

Niels Nes and Martin Kersten

University of Amsterdam
{niels,mk}@wins.uva.nl

Abstract. Content-based image retrieval systems rely on a query-by-example technique often using a limited set of global image features. This leads to a rather coarse-grain approach to locate images. The next step is to concentrate on queries over spatial relations amongst objects within the images. This calls for a small collection of image retrieval primitives to form the basis of an image retrieval system. The Acoi algebra is such an extensible framework built on the relational algebra. New primitives can be added readily, including user-defined metric functions for searching. We illustrate the expressive power of the query scheme using a concise functional benchmark for querying image databases.

keywords: Image Retrieval, Query Algebra, Features, and IDB.

1 Introduction

With the advent of large image databases becoming readily available for inspection and browsing, it becomes mandatory to improve image database query support beyond the classical textual annotation and domain specific solutions, e.g. [21]. An ideal image DBMS would provide a data model to describe the image domain features, a general technique to segment images into meaningful units and provide a query language to study domain specific algorithms with respect to their precision and recall capabilities. However, it is still largely unknown how to construct such a generic image database system.

Prototype image database systems, such as QBIC [6], WebSeek[18], and PictoSeek [8] have demonstrated some success in supporting domain-independent queries using global image properties. Their approach to query formulation (after restricting the search using textual categories) is based on presenting a small sample of random images taken from the target set and to enable the user to express the query (Q_1) "find me images similar to this one" by clicking one of the images provided. Subsequently the DBMS locates all images using its built-in metrics over the global color distribution, texture, or shape sets maintained.

However, this evaluation technique is bound to fail in the long run for several reasons. First, random sample sets to steer the query process works under the assumption that there is a clear relationship between color, texture and shape and the semantic meaning. This pre-supposes rather small topical image databases

and fails when the database becomes large or filled from many sources, such as envisioned for the Acoi image database[1].

Second, the global image properties alone are not sufficient to prune false hits. Image databases are rarely used to answer the query Q_1. Instead, the intended user query (Q_2) is :"find me an image that contains (part of) the one selected" where the containment relationship is expressed as a (predefined) metric over selected spatial and image features or directly (Q_3) :" find me an image that contains specific features or objects using my own metric". In addition to color distribution and texture, spatial information about object locality embedded in the image is needed. A prototype system that addresses these issues is VisualSEEK[18] graphical user interface for the WebSeek image retrieval system.

What are the necessary primitives to express the metric? For example, in a large image database one could be interested to locate all images that contain part of the Coca-Cola logo. This query could be formulated by clipping part of a sample Coca-Cola logo to derive its reddish (R) and white (W) color and to formulate a query of the form:

select display(i)
from image_region r1,r2, image i
where distance(r1.avghue, R) < 0.2
and distance(r2.avghue, W) < 0.2
and r1 **overlaps** r2
and r1,r2 **in** i
sort by distance(r1.avghue, R), distance(r2.avghue, W)

This query uses two primitive parameterized metric functions. The function *distance* calculates a distance in the color space and *overlaps* determines region containment. The former is defined as part of the **color** data type and the latter for the **region** data type.

A challenge for image database designers is to identify the minimal set of features, topological operators, and indexing structures to accommodate such image retrieval queries. In particular, those (indexed) features where their derivation from the source image is time consuming, but still can be pre-calculated and kept at reasonable storage cost. This problem becomes even more acute when the envisioned database is to contain over a million images.

In [13] we introduced an extensible image indexing algorithm based on rectangular segmentation of regions. Regions are formed using similarity measures. In this paper we extended this approach with a query algebra to express queries over the image database.

For such an image retrieval algebra we see three global requirements.

1. The algebra should be based on an extensional relational framework.
2. The algebra should support proximity queries and the computational approach should be configurable by the user.
3. The algebra should be computationally complete to satisfy the wide community of (none-database) image users.

[1] Acoi is the experimental base for the national project on multi-media indexing and search (SION-AMIS project), http://www.cwi.nl/~acoi/Amis

The remainder of this report is organized as follows. In Section 2 we explain our database model, review available region representations and query primitives for image retrieval systems. Also we provide a short introduction to our underlying database system, called Monet. Section 3 explains the query primitives. In Section 4 we define the Acoi Image Retrieval Benchmark, which is the basis for the experimentation reported in section 5. We conclude with an indication of future research topics.

2 Image Databases

This Section introduces the data model for query formulation. The data model is based on regions as the abstraction of the image segmentation process. In section 2.2 we review several region representation methods. In section 2.3 query language extensions for image retrieval are reviewed to provide the background information and to identify the requirements imposed on the image DBMS. Since our image retrieval system is built using the Monet database system we also give a short introduction to Monet.

2.1 Image Database Model

The Acoi database is described by the ODL specification shown in Figure 1.

```
interface Img {
        relationship set < Pix > data inverse Pix::image;
};
interface Pix {
        relationship Img image inverse Img::data;
        relationship Reg region inverse Reg::pixels;
};
interface Reg {
        relationship set < Pix > pixels inverse Pix::region;
        relationship set < Seg > segments inverse Seg::regions;
};
interface Seg {
        relationship set < Reg > regions inverse Reg::segment;
        relationship set < Obj > object inverse Obj::segments;
};
interface Obj {
        relationship set < Seg > segments inverse Seg::object;
};
```

Fig. 1. Data Model

The *data* relationship relates raw pixel information with an image. This is a virtual class, because each pixel is accessed from the image representation upon

need. The *region* relationship expresses that each pixel is part of one region only. For the time being we use rectangular regions to simplify implementation and to improve performance. The architecture has been set up to accommandate other (ir-) regular regions, like hexagons and triangulation, as well.

The *segments* mapping combines regions into meaningful units. The segments are typically the result of the image segmentation process, which determines when regions from a semantic view should be considered together. The model does not prescribe that regions are uniquely allocated to segments. A region could be part of several segments and applying different segmentation algorithms may result in identical region sets. The segmentation algorithm used for the current study is based on glueing together regions based on their average color similarity and physical adjacency, details of which can be found in [13].

The relationship object of the segments interface, expresses that segments can form a semantically meaningful object. An example is a set of segments together representing a car.

2.2 Segment Representation

The bulk of the storage deals with region representation, for which many different approaches exist. All have proven to be useful in a specific context, but none is globally perfect. The chain code as described by Freeman [7] encodes the contour of a region using the 8-connected neighborhood directions. Chain codes are used in edge, curve and corner finding algorithms [11]. It is not useful for region feature extraction, since it only represents part of the boundary of an area, no interior. The complexity is $O(p)$ for both storage and performance, where p is the perimeter of the region.

Many boundary representations exist [10], e.g. polygons and functional shape descriptors. Functional shape descriptors use a function to approximate the region boundary. Fourier, fractal and wavelet analysis have been proposed for this [3, 12, 17]. Although these representations have very low storage requirements, i.e. each boundary is represented using a few parameters, they are of limited use aside from shape representation. Recalculation of the regions interior from polygons is very hard and from functional descriptions generally impossible.

Another representation to describe the interior of the region is run length encoding using (position, length) pairs in the scan direction [9]. Diagonal shaped regions are handled poorly by this coding schema.

The pyramid structures [20, 19] represent an region using multiple levels of detail. They are used in image segmentation and object recognition [20, 15]. These structures are very similar to the quad tree [16]. The quad tree is a hierarchical representation, which divides regions recursively into four equal sized elements. The complexity of this structure per region is $O(p + n)$, where the region is located in a $2^n * 2^n$ image and p is again the perimeter of the region. Quad trees can be stored efficiently using a pointerless representation. The quad tree has been used to represent binary images efficiently. The tree needs only to store those regions which have a different color than its parent nodes.

Since none of the structures above solve the regions representation problem, there is a strong need for an extensible framework. It would permit domain specific representations to be integrated into a database kernel, such that scalable image databases and their querying becomes feasible.

To explore this route we use a minimalistic approach, i.e. regions are described by rectangular grids. The underlying DBMS can deal with them in an efficient manner. The domain specific rules and heuristics are initially handled by the query language and its optimizer.

2.3 Image Retrieval Algebra

The image retrieval problem is a special case of the general problem of object recognition. When objects can be automatically recognized and condensed into semantic object descriptors, the image retrieval problem becomes trivial. Unfortunately, object recognition is only solved for limited domains. This calls for an image feature database and a query algebra in which a user can express domain specific knowledge to recognize the objects of interest.

Research on image retrieval algebras has so far been rather limited. The running image retrieval systems support query by example[6] or by sketch [18], only. For example, the interface of the QBIC system lets the user choose for retrieval based on keywords or image features. These systems have a canned query for which only a few parameters can be adjusted. It does not provide a functional or algebraic abstraction to enable the user to formulate a specific request. In the WebSeek Internet demo the user can adjust a color histogram of a sample image to specify the more important colors. However, this interface allows no user defined metric on colors.

Only Photobook [14] allows for user defined similarity metric functions through dynamically loadable C-libraries. Although this approach is a step forward, it is still far from a concise algebraic framework that has boosted database systems in the administrative domain. In section 3 we introduce the components of such an algebra.

2.4 Extensible Database Systems

Our implementation efforts to realize an image database system are focussed on Monet. Monet has been designed as a next generation system, anticipating market trends in database server technology. It relies on a network of workstations with affordable large main memories (> 128 MB) per processor and high-performance processors (> 50 MIPS). These hardware trends pose new rules to computer software – and to database systems – as to what algorithms are efficient. Another trend has been the evolution of operating system functionality towards micro-kernels, i.e. those that make part of the Operating System functionality accessible to customized applications.

Given this background, Monet was designed along the following ideas:

- *Binary relation storage model.* Monet vertically partitions all multi-attribute relationships in Binary Association Tables (BATs), consisting of `[OID,attribute]` pairs. This Decomposed Storage Model (DSM) [5] facilitates table evolution. And it provides a canonical representation for a variety of data models, including an object-oriented model [1]. Moreover, it leads to a simplified database kernel implementation, which enables readily inclusion of additional data types, storage representations, and search accelerators.
- *Main memory algorithms.* Monet makes aggressive use of main memory by assuming that the database hot-set fits into its main memory. For large databases, Monet relies on virtual memory management by mapping files into it. This way Monet avoids introducing code to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it gives advice to the lower level OS-primitives on the intended behavior[2] and lets the MMU do the job in hardware. Experiments in the area of Geographical Information Systems[2] and large object-oriented applications [1] have confirmed that this approach is performance-wise justified.
- *Monet's extensible algebra.* Monet's Interface Language (MIL) is an interpreted algebraic language to manipulate the BATs. In line with extensible database systems, such as Postgres, Jasmine and Starburst, Monet provides a Monet Extension Language (MEL). MEL allows you to specify extension modules to contain specifications of new atomic types, new instance- or set-primitives and new search accelerators. Implementations have to be supplied in C/C++ compliant object code.

3 Algebraic Primitives

Analysis of the requirements encountered in image retrieval and the techniques applied in prototype image systems, such as [6, 18, 8], indicate the need for algebraic operators listed in Table 1. The parameter i denotes an image, p a pixel, r a region, s a segment and o an object. Most functions are overloaded as indicated by a combination of *iprso*.

The first group provides access to the basic features of images, pixels, regions, segments and objects. Their value is either stored or calculated upon need. The Point, Color, Vector and Histogram datatypes are sufficient extensions to the base types supported by the database management system to accommodate the features encountered in practice so far.

The second group defines topological relationships. This set is taken from [4], because there is no fundamental difference between spatial information derived from images and spatial information derived from geographic information systems.

The third group addresses the prime algorithmic steps encountered in algorithms developed in the Image processing community. They have been generalized from the instance-at-a-time behavior to the more convenient set-at-a-time

[2] This functionality is achieved with e.g. `mmap()`, `madvise()`,and `mlock()` Unix system calls.

Properties	
$area(iprso)$	$\rightarrow float$
$perimeter(iprso)$	$\rightarrow float$
$center(iprso)$	$\rightarrow point$
$avg\_color(iprso)$	$\rightarrow color$
$color\_hist(iprso)$	$\rightarrow Histogram$
$texture(iprso)$	$\rightarrow vector$
$moment(iprso)$	$\rightarrow float$
Topological operations	
$touch(prso, prso)$	$\rightarrow boolean$
$inside(prso, prso)$	$\rightarrow boolean$
$cross(prso, prso)$	$\rightarrow boolean$
$overlap(prso, prso)$	$\rightarrow boolean$
$disjoint(prso, prso)$	$\rightarrow boolean$
Join operations	
$F\_join_{f(prso,prso)}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
$M\_join_{d(prso,prso),m}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
$P\_join_{p(prso,prso)}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
Selection operations	
$F\_find_{f(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow iprso$
$M\_select_{d(iprso,iprso),m}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
$P\_select_{p(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
Ranking and Sample operations	
$P\_sort(\{iprso\})$	$\rightarrow \{iprso\}$
$M\_sort_{d(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
$N\_sort(\{iprso\})$	$\rightarrow \{iprso\}$
$Top(\{iprso\}, int)$	$\rightarrow \{iprso\}$
$Slice(\{iprso\}, int, int)$	$\rightarrow \{iprso\}$
$Sample(\{iprso\}, int)$	$\rightarrow \{iprso\}$

Table 1. The Image Retrieval Algebra

behavior in the database context. This group differs from traditional relational algebra in stressing the need for θ-like joins and predicates described by complex mathematical formulae.

A *fitness* join (F_join) combines region pairs maximizing a fitness function, $f(rs, rs) \rightarrow float$. The pairs found merge into a single segment. The *metric* join (M_join) finds all pairs for which the distance is less than the given maximum m. The distance is calculated using a given metric function, $d(rs, rs) \rightarrow float$. The last function in this group, called *predicate* join (P_join), is a normal join which merges regions for which the predicate p holds. An example of such an expression is the predicate "similar", which holds if regions r_1 and r_2 touch and the average colors are no more than 0.1 apart in the domain of the color space. A functional description is:

$$\begin{aligned} &\text{similar}(r\ r_1, r\ r_2) := \\ &\qquad touch(r_1, r_2) \text{ and} \\ &\qquad distance(r_1.avg\_color, r_2.avg\_color) < 0.1 \end{aligned}$$

The next group of primitives is needed for selection. The fitness find (F_find) returns the region which fits best to the given region, according to fitness function $f(rs, rs)$. The metric select (M_select) returns a set of regions at most at distance m, using the given metric $d(rs, rs)$ function. The predicate select (P_select) selects all regions from the input set for which the predicate is valid.

Join operations	result
$F\_join_f(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR, \nexists l'r' \in LR \wedge f(l',r') > f(l,r)\}$
$M\_join_{d,m}(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR \wedge d(l,r) < m\}$
$P\_join_p(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR \wedge p(l,r)\}$
Selection operations	**result**
$F\_find_f(L,r) \rightarrow iprso$	$l \in L, \nexists l' \in L \wedge f(l',r) > f(l,r)$
$M\_select_{d,m}(L,r) \rightarrow \{iprso\}$	$\{l \mid l \in L \wedge d(l,r) < m\}$
$P\_select_p(L,r) \rightarrow \{iprso\}$	$\{l \mid l \in L \wedge p(l,r)\}$

Table 2. Signatures of the Join and Selection operations

The last group can be used to sort region sets. We have encountered many algorithms with a need for a partial order. P_sort derives a partial order amongst objects. Each entry may come with a weight which can be used by the *metric* sort (M_sort). This sort operation is based on a distance metric between all regions in the set and a given region. The N_sort uses a function to map regions onto the domain $\mathcal{N}$.

After the partial order the Top returns the top n objects of the ordered table. The *Slice* primitive will slice a part out of such an ordered table. The *Sample* primitive returns a random sample from the input set.

4 Acoi Image Retrieval Benchmark

The next step taken was to formulate a functional benchmark of image retrieval problems. Many such performance benchmarks exist for DBMS for a variety of application areas. Examples in transaction processing are the TP series (TPC-C and TPC-D) and in geographic information systems the SEQUOIA 2000 storage benchmark. We are not aware of similar benchmarks for image retrieval. The construction of such a benchmark is one of the goals of Amis. Both the database and image processing community would benefit from such a public accessible benchmark.[3]

Its function is to demonstrate and support research in image processing and analysis in a database context. Therefore, we derived the following characteristics from the algorithms used in the image processing domain.

[3] Readers can contact authors for a copy of the Acoi Benchmark.

- *Large Data Objects* The algorithms use large data objects. Both in terms of base storage (pixels), but also the derived data incurs large space overhead.
- *Complex Data Types* The algorithms use specialized complex data types. Derived data is often stored in special data structures.
- *Fuzzy data* The computational model used is based on heuristics and fuzzy data. This fuzzy data should be accompanied by some form of fuzzy logic.

The Acoi Benchmark Data The data for the benchmark consists of two Image sets, one of 1K images and one of 1M images. The images are retrieved randomly from the Internet using a Web robot. The set contains all kinds of images, i.e. binary and gray scale, small and large but mostly color images.

The Acoi Benchmark Queries Based on the characteristics encountered in the image processing community a set of 6 distinctive queries for the benchmark was identified, which are shown in Table 3.

nr	query
Q1	DB-load
Q2	$\{h \mid i \in Ims \wedge h = normalized\_color\_histogram(i)\}$
Q3	$\{i \mid i \in Ims \wedge L^2 distance(normalized\_color\_histogram(i), h) < 0.1\}$
Q3a	sort Q3
Q4	$\{n_1 n_2 \mid n_1 n_2 \in Regs(im) \wedge \nexists n3 \in Regs f(n_1, n_3) > f(n_1, n_2)\}$
Q5	$\{rs \mid rs \subset Regs(i) \wedge \forall r_1 r_2 \in RS :$ $L^2 distance(avg\_color(r_1), avg\_color(r_2)) < 0.1$ $\wedge \exists s_0 \ldots s_n \in rs :$ r touch $s_0 \wedge$ s_i touch $s_{i+1} \wedge$ s_n touch $s\}$
Q6	$\{i \mid \forall s_i \in Q6(i) \exists s_e \in Segs(e)$ $d(s_i, s_e) < min\_dist\}$
Q6a	sort Q6

Table 3. Benchmark Queries

Query 1 loads the database DB from external storage. This means storing images in database format and calculation of derived data. Since the benchmark involves both global and local image features this query may also segment the images and pre-calculate local image features.

Query 2 is an example of global feature extraction as used in QBIC. This query extracts a normalized color histogram. We only use the Hue component of the Hue, Saturation, Intensity color model. The histogram has a fixed number of 64 bins. In query 3 these histograms are used to retrieve histograms within a given distance and the related images. The histogram h should have 16 none-zero bins and 48 zero. The none-zero bins should be distributed homogeneous over the histogram. The query Q3a sorts the resulting set for inspection.

Query 4 finds the nearest neighboring regions in an image. Near is defined here using a user-defined function, f. This function should be chosen so that neighbors touch and that the colors are as close as possible.

Query 5 segments an input image. Segmentation can also be done with specialized image processing functions, but to show the expressive power of the algebra we also include it here in its bare form. Finally Q6 searches for all images in the database which have similar segments as the example image. The resulting list of images is sorted in query 6a.

The Benchmark Evaluation To compare the results of various implementations of the benchmark we used the following simple overall evaluation scheme. The performance of the Acoi Benchmark against different implementation strategies can be compared using the sum of all query execution times. This way moving a lot of pre-calculation to the DB-load query will not improve performance unless the information stored has low storage overhead and is expensive to recalculate on the fly.

5 Performance Assessment

The benchmark has been implemented in Monet using its extensible features. Details about Monet can be found at {http://www.cwi.nl/~monet}. The DB-load query loads the images using the image import statement into the Acoi_Images set. We only load the images in the system. No pre-calculation has been performed.

The color histogram query (Q2) can be expressed in the Acoi algebra as follows:

```
var Q2 := [normalized_color_histogram](Acoi_Images);
```

The brackets will perform the operation normalized_color_histogram on all images int the Acoi_Images set. It returns a set of a histograms. Q3 uses a M_select with the L^2 metric. The sorting of Q3a can be done using the M_sort primitive. Query Q4 is implemented in the Acoi algebra using a F_join with the function $f(r_1, r_2)$ defined as follows:

$f(r_1, R_2) :=$
dist($r_1.color()$, $r_2.color()$) if r_1.touch(r_2)
max_dist

The queries 5 and 6 are implemented by longer pieces of Monet code. The segmentation of query Q5 use an iterative process. This process can make use of the F_join primitive to find the best touching regions based on the color distance, see [13] for full details.

Query Q6 can be solved using a series of M_select calls. For each segment in the example image we should select all Images with similar segments, where similar is defined using the metric given. The intersection of the selected images is the result of query 6. This can be sorted using the M_sort primitive.

The Benchmark Results We run these queries using the small Acoi database of 1K images. The small benchmark fits in main memory of a large workstation. The database size is approximately 1G. We used a sparc ultra II with 128 MB of

main memory running the Solaris operating system, to perform the benchmark on. Using the Acoi algebra we could implement the benchmark with very little effort.

The initial results can be found in Table 5. Overall the benchmark took 3468.8 seconds.

In the result we can see that the DB-load query takes more than 80 percent of the overall benchmark result. This unexpected result stems from heavy swapping of the virtual memory management system. Main memory runs out quickly, so swapping will influence the performance. Based on our early experimentation with multi-giga-byte databases this problem can be resolved with some careful loading scripts.

We found that the results of queries Q4 and Q5 were low. The none-optimized current implementation of F_join was responsible for the low performance. To improve it we moved the spatial constrains out of the F_join. This allows us to find candidate pairs based on the spatial relation between regions quickly. This way we improved the performance of the queries Q4 from 5 to 1 second and Q5 from 21 to 1.2 seconds using a few minutes of programming. A similar step in a traditional image programming environment would have meant partly re-coding several pages of c/c++ code.

Query	Time(s)	Query	Time(s)
Q1	2865	Q4	1.0
Q2	598	Q5	1.2
Q3	1.5	Q6	1.5
Q3a	0.3	Q6a	0.3

Table 4. The Acoi Benchmark Results

6 Conclusions

In this paper we introduced an algebraic framework to express queries on images, pixels, regions, segments and objects. We showed the expressive power of the Acoi algebra using a representative set of queries in the image retrieval domain. The algebra allows for user defined metric functions and similarity functions, which can be used to join, select and sort regions. The algebra is extensible with new region properties to accommodate end user driven image analysis in a database context.

We have implemented the algebra within an extensible DBMS and developed a functional benchmark to assess its performance. In the near future we expect further improvement using extensibility in search methods and index structures to improve the performance of the algebra. As soon as the full Acoi database is ready we will perform the benchmark on the set of 1M images.

References

1. P.A. Boncz and M.L. Kwakkel, F. Kersten. High Performance support for OO traversals in Monet. In *BNCOD proceedings*, 1996.
2. Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its Geographic Extensions: a novel Approach to High Performance GIS Processing. In *EDBT proceedings*, 1996.
3. R. Chellappa and R. Bagdazian. Fourier Coding of Image Boundaries. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:102–105, 1984.
4. E. Clementini, P. Felice Di, and P. Oosterom van. A Small Set of Formal Topological Relationships Suitable for End-user Interaction. In *SSD: Advances in Spatial Databases*. LNCS, Springer-Verlag, 1993.
5. G. Copeland and S. Khoshafian. A Decomposed Storage Model. In *Proc. ACM SIGMOD Conf.*, page 268, Austin, TX, May 1985.
6. C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *Intelligent Information Systems 3*, pages 231–262, 1994.
7. H. Freeman. On the encoding of arbitrary geometric configurations. *Transactions on electronic computers*, 10:260–268, jun 1961.
8. T. Gevers and A. W. M. Smeulders. Evaluating Color and Shape Invariant Image Indexing for Consumer Photography. In *Proc. of the First International Conference on Visual Information Systems*, pages 293–302, 1996.
9. S W Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory 12(3)*, pages 399–401, july 1966.
10. Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1989.
11. Hong-Chih Liu and M. D Srinath. Corner Detection from Chain-Code. *Pattern Recognition(1-2), 1990*, 23:51–68, 1990.
12. B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Co., New York, rev 1983.
13. N.J. Nes and M.L. Kersten. Region-based indexing in an image database. *In proceedings of The International Conference on Imaging Science, Systems, and Technology, Las Vegas*, pages 207–215, June 1997.
14. A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. In *SPIE Storage and Retrieval for Image and Video Databases II, No. 2185*, pages 34–47, 1994.
15. E. M. Riseman and M. A. Arbib. Computational Techniques in the Visual Segmentation of Static Scenes. *Computer Graphics and Image Processing*, 6(3):221–276, June 1977.
16. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990.
17. J. Segman and Y. Y. Zeevi. Spherical wavelets and their applications to image representation. *Journal of Visual Communication and Image Representation*, 4(3):263–70, 1993.
18. John R. Smith and Shih-Fu Chang. Tools and Techniques for Color Image Retrieval. In *SPIE Storage and Retrieval for Image and Video Databases IV, No 2670*, 1996.
19. S. L. Tanimoto and T. Pavlidis. A Hierarchical Data Structure for Picture Processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.
20. L. Uhr. Layered recognition cone networks that preprocess, classify, and describe. *IEEE Transactions on Computers*, 21:758–768, 1972.
21. Aref. W.G., Barbara D., and D. Lopresti. Ink as a First-Class Datatype in Multimedia Databases. Multimedia Database Systems, pages 113–160, 1996.

A Meta-Structure for Supporting Multimedia Editing in Object-Oriented Databases *

Greg Speegle[1] and Xiaojun Wang[2] and Le Gruenwald[3]

[1] Baylor University, Waco, TX 76798, USA
[2] Intervoice, Inc, Dallas TX 75252, USA
[3] The University of Oklahoma Norman, OK 73019, USA

Abstract. Multimedia databases include many types of new data. One common property of these data items is that they are very large. We exploit the concept of transformational representations to logically store images without the bitmap. This technique is similar to views in traditional database systems. The technique is useful when users are editing images in the database to create new images. Our method produces significant space savings over *already compressed* images, and potentially greater savings for video and audio. The method requires the support of multimedia editors. This paper emphasizes the meta-structure needed by the database to support multimedia editing.

1 Introduction

Multimedia has become the quintessential element in computer applications. Programs are purchased on CDROMs loaded with graphics, audio, animations, and videos. Media viewers are common on home computers. Millions of users are putting digital media on the web.

As the demand for multimedia continues to increase, computer systems will have to address two large problems associated with the wide spread use of digital media. The first is that current systems simply store such media as compressed files on disks without providing any organization. This, therefore, requires a tremendous amount of effort from application programmers to interpret and manipulate those files. The second is that even with advanced compression techniques and several gigabyte disks being common, the size of media data causes it to consume enormous amounts of space. For example, a 75 minute CD may use 100 MB of storage, and each frame of a video may use over 1 MB [8]. This problem is increased when users edit digital media in order to refine their presentations.

Editing can occur in video games (graphics), recording studios (sound effects), and video-on-demand systems (editing for content). In each of these examples, the application must maintain many large multimedia objects that are similar. This causes a tremendous amount of replicated information to be stored. This situation would arise in any multimedia authoring application in which

* Work partially supported by Baylor University Grant 009-F95

users will want to save each version of a file. The storage cost of such redundant information can vary from 2KB for a small graphic to over a gigabyte for a movie. Without a technique to efficiently store the redundant data, the systems supporting such applications would quickly run out of memory.

We propose an object-oriented MultiMedia Database Management System (MMDBMS) to solve the problems of organization and space consumption. Instead of storing the objects themselves, our MMDBMS stores the editing operations, called the transformational representations, used to create media objects. The collection of all information used to make a new image is called a *transform*. Since the editing operations would only require a few bytes of space, transforms would significantly reduce storage requirements for already compressed data [11, 12, 1].

Transformational representation of information is called a *view* in relational databases, and an *intensional database relation* in logical models like datalog [14]. Within media editing tools, transformational representations of objects are also used. For example, some editors, such as JPEGview [3], can store an object by copying the original file and then appending to it a series of commands which indicate how the edited image should be recreated. Unfortunately, using application specific transformational representations does not allow an image to be correctly viewed by another editor. This happens because the editing commands used by one system are meaningless to another. Likewise, this method has the disadvantage of storing redundant data, since the original file is copied to the edited one. Although in a file system this redundancy is unavoidable – since it is impossible to prevent the deletion or movement of files – in a database, actions can be performed as part of a deletion operation in order to protect the derived data. One simple solution is to instantiate the derived data before allowing the deletion to complete. Thus, we do not need the redundant storage. Furthermore, image degradation due to repeated application of lossy compression algorithms is eliminated. Thus, better image quality and greater space efficiency can be gained.

Ideas similar to multimedia transforms can be found in [2] and [13]. In [2], multimedia objects are treated as binary large objects (BLOBs). Editing of media objects is translated into operations on BLOBs. Handles to BLOBs are stored in a relational database. Unfortunately, BLOB operations are not sufficient to perform all required media object operations. For example, there is no way to describe the dithering needed to double the size of an image. In [13], a notion similar to transforms is presented in which scripts are passed to image editing tools in order to create new images. However, their approach performs these operations outside the database itself. This can lead to problems with deleted references and poor specifications, both of which can be solved by including the new images within the database.

Within the OODBMS literature, a similar notion to our approach is the concept of versioning [9]. Versions allow modifications to an object without losing the previous state. This is accomplished by having additional structures, similar in nature to those proposed here, which keep up with the history of the object.

However, there is a fundamental difference between versions and our work. In our system, each modification creates a totally independent object, not merely a copy of another object. Thus, the support structure for our work must be different in terms of modifications to the database.

The rest of the paper is organized as follows. Section 2 presents the basic model used by our meta-structure. Section 3 presents implementation details for capturing the operations, storing them in the database, and instantiating the transforms. Section 4 concludes this work and outlines the rest of our project. It should be noted that this paper focuses on images. However, we intend to apply this meta-structure to all media types.

2 The Basic Model

Our base model for an MMDBMS with transforms contains two distinct parts. The first part, which is the emphasis of this paper, is the meta-structure needed to support transforms. This is covered in Section 2.1. A second needed component involves the language used to define the operations for the media. Our language is only a small component of this paper, and is simply used to show that such a language is possible. Our proposed language is in Section 2.2.

2.1 Meta-Structure for Views

Our MMDBMS assumes the existence of a media hierarchy – a set of classes which describe the multimedia data. At database design time, the DBA determines if a class is *transformable*. This is a property of the class, and as such it is inherited by its subclasses. If a class is transformable, then by default, all updates to objects in the class will result in the specification of objects, and not create new media objects to be stored in the database. Thus, each of these virtual objects will have a transformational representation of how the object can be recreated from objects physically stored in the database, but they will not include the binary representation of the object.

Note that this implies that specifications are defined on objects, rather than on classes. In traditional databases, views defined like this would be of little benefit, but the extremely large size of individual objects in multimedia systems allows this approach to reduce storage costs. However, using objects as the basis for transforms does require that the system have additional information in order to keep up with the transforms. This additional information is the basis of our meta-structure. We create a *transform-dag* which represents the history of the transform. An arc in a transform-dag from o_1 to o_2 means that o_1 is used to create o_2. In this case, o_1 is called the parent, and o_2 is called the child. As its name suggests, a transform-dag is a directed acyclic graph. The roots of every transform-dag are stored objects. Every transform is a member of exactly one transform-dag. We consider a stored object to be the root of many dags if the dags contain no common transforms. Likewise, each class may have many transform-dags, and a transform-dag may span classes.

Once a class has been defined as transformable, a new system defined attribute is added to the class to support transform-dags. Although it would have a unique system defined name, we use the attribute name *children* in this paper. This attribute would have the type *set-of oids*, where each oid is a transform, and would be a child in the transform-dag.

The system must also create a special *transform-of*-**class** for each transformable class. The *transform-of*-**class** is semi-automatically generated. The *transform-of*-**class** contains exactly the same attributes and methods as the base class, including the children attribute added by the system, except for the following changes:

1. A system defined attribute *parent* is added. The parent is also a *set-of oids*, and represents the objects that were used to create an instance of the *transform-of*-**class**.
2. Two specifications are added. The short specification defines the creation of an object from its parents. The long specification defines the creation of an object from stored objects.
3. A state attribute is added. This is also a system defined attribute which is used to determine the status of transforms. A transform may be either Permanent–meaning a user has saved it; Stable–meaning the system has temporarily saved it; or Transient–meaning the objects can can have their media content modified.
4. The binary representation attribute is removed. This is the part which is semi-automatic, since there is no standard way to describe this field. The binary representation is the physical data which defines a stored object.

An example of a **jpeg** class and a *transform-of*-**jpeg** class is in Figure 1. Note that the *transform-of*-**class** is not a superclass of the **jpeg** class since the specifications are not inherited, nor is it a subclass since the binary representation is no longer present.

Both specifications are linked lists of *Specification Entries*, denoted SE. An SE consists of two parts. The first part is a text description of the command issued by the user. The description consists of two components. The first component is the actual model language command issued, and the second is the parameter required by that command. The second part of the SE is called the *reference* and is used to structure the SE. The reference also consists of two parts, the base and the offset. The base is the object id of a *stored* media object which is one of the roots of the transform-dag of the transform. The offset is an integer value which indicates what changes (if any) have been done to the base object. A negative value (usually -1) indicates that the object has not been edited, and the base object must be retrieved from the database. A value of 0 indicates that the base object has been modified, and the desired image is the result of the previous SE. A positive value is the object id of the SE which represents all of the changes to the base object. Note that the use of 0 is redundant since the oid of the previous SE could also be used. However, this condition is so common, that using a special value to represent it allows an easy optimization of holding the previous SE in memory so that it can be accessed directly.

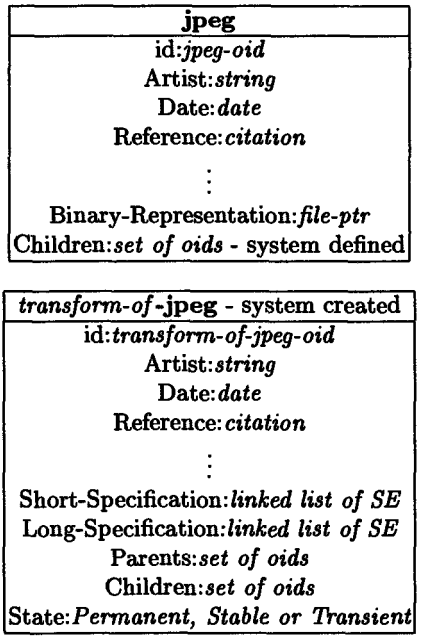


Fig. 1. Example Classes

In order for an MMDBMS to exploit transforms, it must be the case that the editing tools support transforms as well. Some editing tools already use this concept in order to prevent image degradation, but they must go a step further in order to interface with the MMDBMS. The editing tools must be able to transmit to the database system the operations performed by the user. Furthermore, these operations must comply with the logical model language (LML) of the database. An example LML is presented in Section 2.2. It is not practical for the database system to translate proprietary commands into the model language, simply because there are too many different implementations possible. Likewise, the editing tool must transmit to the database system additional operations to retrieve, store, modify and delete images. This is similar to tools accessing an SQL database today. Once the formal model language is established, then it is reasonable for tools to comply with a standard. As image processing continues to mature, such a standard is very likely to emerge, and may even evolve from current standards being developed for handling derived video information, such as MPEG-4 [6].

However, even if media editing matures to the point that this type of functionality can be achieved, a MMDBMS which uses transforms has to fulfill many requirements. Such a system must be able to instantiate objects from formal

model specifications, and provide the media to editors on demand. It must be able to accept commands from editors and integrate them into specifications. Finally, it must maintain the transform-dag with media objects being added and being deleted from the system. Section 3 provides details on how these requirements can be fulfilled.

2.2 Model Language for Images

In order for transforms to be defined on multimedia objects, a standard LML must be developed. We predict this language to evolve once editing tools mature. However, in order to continue with our work, we propose a simple image modeling language which captures many of the transformations performed by users with editing tools. Further work is in progress to determine a complete, minimal and independent set of operations for image editing [1].

Our model language for images is based on the image algebra defined in [10]. Ritter's Image Algebra is a mathematical definition of operations used to process and analyze images. As such, it contains a large number of complex operations. This allows the image algebra to capture not only operations performed on images, but also image recognition operations like edge detection.

Our goal is to define a simple set of operations which still capture all of the editing commands performed by users. Thus, for this work we use the five operations: Merge, Define, Mutate, Modify and Combine. These operations form a natural set of basic operations for image manipulations in systems which do not add information to images by techniques such as drawing. Drawing, manipulating the color palette and channel operations are needed in a complete set of image editing tools. Finally, this restricted model is designed only for RGB (red-green-blue) color models.

The Merge operation combines two images together. One image is the base, and the other is the new information. The new information is merged into the base. The Merge operation allows for several different types of combinations. For example, the merge can be transparent, meaning that the base image is displayed wherever the new image is showing its background color, or it can be opaque, meaning that the background information of the new image is also pasted into the base image. The Merge operation works even if the images are not the same size or shape. If the new image overlaps an area not originally part of the base image, the result is simply extended to include the new area.

The Define operation is used to define regions within an image. A series of points defines a polygon within an image. The define operation then either adds the space within the polygon to the defined region, or it removes the polygon from the defined region, or it adds (removes) the image NOT within the polygon to the defined region. Within this work, whenever we refer to an image, the region of an image created with the define operation can also be used.

The Mutate operation is used to alter the form of an image. The mutation is accomplished by mapping an image into a form where every pixel is a 5-tuple, (x,y,r,g,b), where (x,y) is the coordinate location of the pixel and (r,g,b) is the color component. Each (x,y) component is put into a 3 x 1 vector (the last

$$\begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta)+y_r(\sin\theta) \\ \sin\theta & \cos\theta & y_r(1-\cos\theta)-x_r(\sin\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

The Mutation Matrix

$$\begin{bmatrix} 0*\cos 90 - 0*\sin 90 + 25*(1-\cos 90) + 25*\sin 90 \\ 0*\sin 90 + 0*\cos 90 + 25*(1-\cos 90) - 25*\sin 90 \\ 0*0+0*0+1*1 \end{bmatrix} = \begin{bmatrix} 50 \\ 0 \\ 1 \end{bmatrix}$$

Fig. 2. Rotation Example – The result of rotating the point (0,0) in a 51 x 51 matrix 90° counter clockwise around the center point – (25,25).

element of which is usually 1). This vector is then multiplied by a 3x3 matrix in order to get a desired effect. An example of a 90° counter-clockwise rotation is in Figure 2. Using interpolation as needed, the resulting image is then mapped back to pixel format, with the order of the pixels defined by their location in the file.

The Modify operation is used to change colors within an image. The Modify operation accepts up to 6 parameters, which correspond to old and new values for the RGB color. A color can be omitted from the operation, and that implies a "don't care" condition for that color. Thus, Modify(Red(60,100)) would modify all pixels having a red value of 60 to a red value of 100, no matter what the values of the green and blue components, while Modify(Red(60,100),Green(60,100), Blue(60,100)) would change only a pixel with 60/60/60 to 100/100/100. Ranges of values can be used for the old value of the pixels. Thus, an entire image can be turned black by Modify(Red(0...255,0),Green(0...255,0), Blue(0...255,0)).

The Combine operation is used to achieve certain effects which are based on localized portions of the image. For example, a blur effect is achieved by making each pixel look a little bit more like the pixels closest to it. Thus, the value of a pixel is computed by taking a weighted average of the pixel and the 8 pixels bordering it. See Figure 3 for an example of a simple blur matrix. Several other effects, such as distort and sharpen, are modeled as combinations of color values.

1	2	1
2	4	2
1	2	1

Weights of pixels

(50,50,50)	(25,50,100)	(50,50,50)
(25,50,100)	(200,200,200)	(25,50,100)
(50,50,50)	(25,50,100)	(50,50,50)

Example original pixels

Fig. 3. Combine Example – Sum of weights divided by 16 and rounded to the nearest integer results in a pixel value for the center of (75, 88, 112).

3 Implementation Issues

In Section 2, several system requirements are listed. In this section, we examine the implementation issues involved with the creation and maintenance of specifications and the upkeep of the transform-dags. Each of these issues is related to the natural uses of a MMDBMS. Instantiation results from a user requesting a transform object. The creation of specifications results from editing operations being applied to opened images. The maintenance of specifications and the upkeep of a transform-dag is a required to support deletion and modification of objects. It should be noted that naive instantiation of images is straightforward processing of the specifications, but *efficient* instantiation is beyond the scope of this paper.

3.1 Creation of Specifications

When an image is requested by an editor, a new object is created. This object is stored in the same *transform-of*-**class** as the requested object, and is a child of the requested transform object. The child's long specification is the same as its parent's, but the child's short specification contains only a reference to the parent. Since the short specification defines the differences between a parent and its child, this short specification implies that the image associated with the child object is identical to the parent's image. When the image is returned to the editor, this child object is prepared to receive the edits performed by the user. The reason for this is that the requested object is *permanent* in the database and may not be changed. This is needed to ensure that other transforms defined on this object will not be invalidated by updates to the permanent object. By contrast, the new object is a *transient* object and can be modified.

After the image is returned to the editor, specifications are added to the new transient object as users edit the image. The editor submits specification texts to the MMDBMS. For most operations, the MMDBMS creates a SE object with the text the same as submitted by the editor, the base being the oid corresponding to the object, and the offset being 0 (the only exception is the first operation which sets the offset to -1). However, merge operations are more complicated. For a merge operation, two SE's are added to the list. The first SE has the "source" of the merge for the base and an offset of -1. The text of the SE is null. The second SE has the destination image as its offset. This offset is the oid of the SE which was at the tail of the specification before the merge operation. The text has the merge command with the parameter being the location of the new information within the destination image. If the source object is transient, it is upgraded to stable to prevent deletion anomalies.

As an extended example, consider the case where images I_1 and I_2 have been edited by a user. In the database, there exists objects O_1 and O_2 which correspond to the results of applying editing operations to the respective base objects. Now assume that the user selects a part of I_2, copies it and pastes it into I_1. When the selection is performed on I_2, the *define* command along with the points defining the region are sent to the database. An SE object, S_1, is

added to the short and long specifications of O_2. S_1 has text set to the define command with its parameters, the offset is set to zero, and the base[4] is set to a default value. However, the copy and paste creates a merge operation, which is more complicated.

With a merge operation, the database first changes the state of O_2 to stable. Next, a new object, O_3, is created as the child of O_2. Any further operations on I_2 will result in updates to the specification of O_3. For further information about the creation of O_3, see Section 3.2. A new SE object, S_2 is added to the short and long specification of O_1. O_2 is added to the parent set of O_1, and O_1 is added to the children of O_2. The text of S_2 is empty, but the base is the object id of O_2 and the offset is -1. A second SE object, S_3, is used to hold the operation. The command of S_3 is *merge*, and the parameter is the location in I_1 for the new information, the base[4] is set to the default value, and the offset is the oid of the SE which defines the destination image. In this case, that would be S_A. A chart of the short specifications is in Figure 4.

O_1				O_2				O_3			
oid:1				**oid:2**				**oid:3**			
SE	command	base	offset	SE	command	base	offset	SE	command	base	offset
S_A	NULL	$I_1.oid$	-1	S_B	NULL	$I_2.oid$	-1	S_C	NULL	2	-1
S_2	NULL	2	-1	S_1	define	0	0				
S_3	merge	0	*oid of* S_A								

Fig. 4. Merge with SE's

3.2 Insertion of New Images

Recall from Section 2 that we assume the editing tool can interface with the MMDBMS. This requires the tool to submit model language commands to the database system, and accept images from the database as described in Section 3.1. However, the editing tool must also interact with the database in order to allow the system to maintain a correct specification of objects. Thus, in addition to the model language commands, the editor must also support additional commands related to storing and retrieving objects and transforms from a MMDBMS. These commands do cause an additional overhead, as the editor must submit them to the database system. However, this overhead is small compared to the effort which must be expended by a user in searching for images in an unstructured environment.

The first such command is **Open**. The purpose of an Open command is to allow the editor to have access to a particular media object. There are many

[4] The base is not used in instantiation when the offset is greater than or equal to zero

levels of interaction which can exist between the editing tool and the database system with respect to this operation. For example, the database could require that the tool creates a query to select the object, or that the tool has a handle or pointer to the object, via browsing through the MMDBMS interface or as a direct association link from another object. Since we are building a simple prototype system, we assume the deepest level of interaction and require the editing tool to pass the actual object id to the database system. In actual practice, this would not be the best system since it would require too much information to be passed to the editor. A more general query or browsing based interface would be easier to implement across multiple MMDBMSs.

Upon receiving the Open command for object O, the database system determines if the class of O is transformable, or if it is a ***transform-of*-class**. If it is not, then the image is from a class which stores the binary representation of all images, and the image is simply returned to the editor and no further work is done. If the editor submits LML commands related to this object, the system ignores them. However, if the class is transformable or a ***transform-of*-class**, then the system creates a new object, O_1, which is the new transform of the current object, O. The object O_1 is stored as a new object in the ***transform-of*-class**. The parent of O_1 is O, and if O is a stored object, then it is the root of a new transform-dag. In the short specification, the first SE for O_1 has a null text field, an offset of -1 and the base is the oid of O. The long specification of O_1 is the same as the long specification of O. The image associated with O_1 is instantiated and returned to the user.

When the user is finished creating a new object, a second new command– **Save** is executed. Since the editing tool does not know if the class is transformable or not, the editor submits the entire object with the save command. If the class is not transformable, the new object is stored in the class, and the additional information, such as creation date and author, is added to the object. If the object is transformable, the object data is discarded, and the transform object, o_1, is now made permanent in the database. As with the non-transformable case, the additional data is also added. Clearly, if the editor is aware that the object is a transform, then significant network bandwidth can be saved by not transmitting the object. However, we are assuming the worst case to show how the database could handle the unwanted data.

If the user terminates the session without saving the object, then the editing tool should send an **Abort** message. When this message is received, the MMDBMS responds as a traditional database would to an abort of a transaction. Uncommitted data is removed from the system. However, in the case of transformable objects, it is certain that uncommitted data exists, and furthermore it must be treated as a deletion (see Section 3.3).

3.3 Deletions and Modifications

We use the principal of ***transform independence*** to resolve deletions and modifications of an image. View independence means that the user should not be aware if the object is a transform or a stored object. This means that deletions

and modifications to an image should not alter any transforms defined on that image. In the case of deletion, the object is removed from the transform-dag. This requires that the children of the object now become children of the deleted object's parents. There are two cases which must be considered here. First, if the deleted image is stored (i.e., not a transform), then its children must now become stored objects since their specifications will be invalidated by the deletion. In order to do this, the children are instantiated as described in Section 3.1. They are then inserted as new objects into the database. The transform-dag is then followed from all of the newly instantiated objects. Each child in the dag must have its long specification rebuilt from the existing short specifications. To rebuild a long specification, the short specification is appended to the end of the long specifications of a transform's parents. The specifications must be re-optimized in order to improve instantiation speed, as any optimizations done could be destroyed in this process. Once this is complete, the transforms corresponding to the instantiated objects are deleted from the system.

The second case is when the deleted object is not stored, and thus has both parents and children. In this case, the long specifications are not changed, but the short specifications of the children must now include information from the deleted parent. Let the deleted parent be O_1, and the child be O. In the short specification of O, there exists an SE object, S_1, with base equal to the object id of O_1 and offset of -1. S_1 is removed from the specification, and the entire short specification of O_1 is put in its place. Furthermore, all references to the oid of S_1 are replaced by the oid of the last SE object in the newly added list. Optimization of the short specification could be performed, but since it is not used to instantiate the object, it is not needed.

Note that if a transient object is not saved in the database, the same process must be followed. Thus, from the extended example in Section 3.1 and Figure 4, if O_2 is not saved in the database, then the short specification of O_2 is inserted in the short specification of O_1 in place of the SE object S_2.

Modifications of images are implemented by treating them as a delete followed by an insertion. Thus, if a stored object is modified, all of the children are instantiated as new stored objects, and the long specifications of all descendants in all transform-dags are updated. The modified object is stored back in the database. If a transform object is modified, it is removed from the transform-dag and the appropriate alterations are made to the short specifications of its former children. However, the modified object is not instantiated. Merely its specification is extended to include the modifications.

4 Conclusion and Future Work

Using transformations to represent images is a powerful, albeit complex, mechanism. A database system which uses transformations can save tremendous amounts of space. Also, transformation based representations do not suffer from image degradation from repeated compressing and decompressing of images.

Thus, such a system provides better images and can store more images in the same amount of space.

However, the complexity of transformational representations carries a cost. The most obvious cost involves the time required to instantiate transformational representations. There is also the additional cost of database support for multimedia editing. Our paper presents a model that can be used to store the media objects using transformational representations. In doing so, we present the meta-structure, model language, and implementation techniques for object instantiation, insertion, deletion, and modification.

We propose the use of a *transform-dag* to represent the history of the creation of an image. An arc from object o_1 to o_2 means that o_1 was used to create o_2. The roots of the dag are the stored objects. Each object has two specifications which are the transformations needed to instantiate the image. One of these specifications is called the *short* specification, and the other is the *long* specification. The short specification indicates how the image would be reconstructed from its parents in the transform-dag. The long specification is used to instantiate the image from the stored objects or the roots of the transform-dag.

Two specifications are required to efficiently handle instantiations and deletions from the database. The long specification would be used to instantiate the image, thus reducing the amount of intermediate work that would be required. However, if only long specifications are used, and the stored object is deleted, then all of the derived objects would have to be instantiated as well. Thus, by allowing the transform-dag to have more than one level, it is possible to instantiate fewer objects on a deletion.

The transformations used here are based on a formal model of image editing derived from the Image Algebra in [10]. The system proposed here assumes that editors translate operations submitted by users into the transformational model, and then send them to the database. A simple LML consisting of five operations – Combine, Define, Merge, Modify and Mutate – is presented.

This work is part of an ongoing project to create a prototype system which can fully exploit transformational representations of images, audio and video. The control logic for the work presented in this paper is available via anonymous ftp from mercury.baylor.edu. The program uses a slightly different version of specification entry objects, but otherwise conforms to the algorithms presented here. In addition to this work, there are several other projects related to integrating transforms within a MMDBMS. First, the image LML must be completed. Issues such as completeness, minimality and independence are explored in [1], but alternative color models and quantization must be added to the model. Second, a tool for instantiating images from the image LML must be designed and implemented. This tool will also be used to instantiate transforms for user requests and when deletions require that the system instantiate images.

Also, it should be noted that the long specification of an object should not be the same as the pure concatenation of the short specifications of its ancestors. The long specifications should be optimized to improve instantiation time. For example, since size manipulations and rotations are accomplished through the

mutate operation, which is matrix multiplication, a rotation and an increase in size can be combined into a single mutate operation in the specification. In order to do optimization, we must understand the compatibility of the operations, which is complicated by the large parameters taken by the commands.

This analysis of the LML commands for optimization also provides an opportunity for exploiting specifications for content-based retrieval. Content-based retrieval is finding an image based on the actual picture, rather than descriptions of it. There is much research in this complex area (see, e.g., [5, 7, 4, 15]). However, the use of specifications provides semantic information not generally available. Our plan is to use a traditional content-based retrieval program on stored images, and then combine those results with the specifications to determine whether or not transforms satisfy the query. As a very simple example, if an image does not have a picture of a red balloon in it, a crop of that image will not either.

Finally, all of the work planned for images is also planned for audio and video. We will develop an LML for both audio and video. We will build editors which can submit operations to a system, and we will be able to instantiate the data from the specifications. We will also optimize the specifications. However, the work done in this paper will not be repeated, as the basic ideas presented here will hold in continuous media.

Thus, the object-oriented meta-structure for the transformational representations of images, is actually the meta-structure for all media types. The use of a dag to indicate the history of an object and a specification to define its form are the basic building blocks of a system which has tremendous capabilities in terms of reducing the amount of space used by multimedia databases, and to improve the quality of the media at the same time.

References

1. L. Brown, L. Gruenwald, and G. Speegle. Testing a set of image processing operations for completeness. In *Proceedings of the 2nd Conference on Multimedia Information Systems*, April 1997. 127-134.
2. J. Cheng, N. Mattos, D. Chamberlin, and L. DeMiciel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994.
3. A. Giles. *JPEGView Help*, 1994. Online help system.
4. Y. Gong. An image database system with content capturing and fast image indexing abilites. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 121–130, May 1994.
5. V. Gudivada and V. Ragshavan, editors. *Computer : Finding the Right Image.* IEEE Computer Society, September 1995.
6. MPEG Integration Group. SO/IEC JTC1/SC29/WG11 coding of moving pictures and audio information, March 1996. N1195 MPEG96/ MPEG-4 SNHC Call For Proposals. ISO.
7. H. Jagadish. *Multimedia Database Systems: Issues and Research Directions*, chapter Indexing for Retrieval by Similarity, pages 165–184. Springer-Verlag, 1996.

8. S. Khoshafian and B. Baker. *Multimedia and Imaging Databases.* Morgan Kaufman, 1996.
9. W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990.
10. G. Ritter. Image algebra, 1995. Available via anonymous ftp from ftp.cis.ufl.edu in /pub/src/ia/documents.
11. G. Speegle. Views as metadata in multimedia databases. *Proceedings of the ACM Multimedia '94 Conference Workshop on Multimedia Database Management Systems*, pages 19–26, October 1994.
12. G. Speegle. Views of media objects in multimedia databases. *Proceedings of the International Workshop on Multi-Media Database Management Systems*, pages 20–29, August 1995.
13. G. Schloss and M. Winblatt. Building temporal structures in a layered multimedia data model. *Proceedings of ACM Multimedia 94*, pages 271–278, October 1994.
14. J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988.
15. A. Yoshitaka, S. Kishida, M. Hirakawa, and T. Ichikawa. Knowledge-assisted content-based retrieval for multimedia databases. *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 131–139, May 1994.

Establishing a Knowledge Base to Assist Integration of Heterogeneous Databases

D. D. Karunaratna[1], W. A. Gray[2], and N. J. Fiddian[2]

[1] Department of Computer Science, University of Colombo, Sri Lanka
[2] Department of Computer Science, University of Wales-Cardiff, UK
{scmddk,wag,njf}@cs.cf.ac.uk

Abstract. In this paper the establishment of a supporting knowledge base for a loosely-coupled federation of heterogeneous databases available over a computer network is described. In such a federation, user information requirements are not static. The changes and evolution in user information needs reflect both the changing information available in the federation as databases join and leave it and their contents change, and the varied requirements the users have of the information available in the federation. As a user's awareness of the contents of the federation improves, their requirements from it often evolve. This means that users need different integrating views if they are to realise its full potential. The knowledge base will be used to help users of the federated databases to create views of the federation and to determine the available information in the federation.

The knowledge base (KB) is created and evolved incrementally by analysing both meta-data of databases as they join the federation and views as they are created over the databases in the federation. The KB is organised as a semantic network of concept clusters, and can be viewed as a conceptual model describing the semantics of all databases in the federation. It records not only concepts and their inter-relationships defined in schemas but also their frequency of occurrence within the federation, and is used both as a semantic dictionary and as a basis to enrich schemas semantically during the detection of semantic relationships among schema components when creating user views. We semantically enrich schemas prior to integration by generating the best sub-network that spans the concepts common to the schema and the knowledge base and then by unifying this sub-network with the schema. Hence the terms in schemas are given interpretations not by considering them in isolation but by taking into account contexts, derived from the knowledge base with respect to those schema terms.

An architecture to facilitate the establishment of this knowledge base and to exploit its functionality is described. The novelty of our research is that we create a knowledge base for a federation by gathering and organising information from it to assist groups of users to create dynamic and flexible views over the databases in the federation which meet their dynamic information requirements. This knowledge base enables re-use of the semantics when creating views.

Keywords : loosely-coupled federation, heterogeneous databases, semantic dictionary, database discovery.

1 Introduction

Recent progress in network and database technologies has changed data processing requirements and capabilities dramatically. Users and application programs are increasingly requiring access to data in an integrated form, from multiple pre-existing databases [6, 34, 42], which are typically autonomous and located on heterogeneous software and hardware platforms. For many of these databases, design documentation, high level representation of domain models (e.g. ER, OMT representations etc.) and additional domain knowledge from the DB designers may not be available. Different users have different needs for integrating data and their requirements may change over time and as new databases become available. This means that the same database may participate in many different ways in multiple integration efforts, hence different views of the databases need to be constructed.

A system that supports operations on multiple databases (possibly autonomous and heterogeneous) is usually referred to as a *multidatabase system* [3, 22, 23, 32]. There are two popular system architectures for multidatabases: tightly-coupled federation and loosely-coupled federation [8, 20, 39]. In a tightly-coupled federation, data is accessed using a global schema(s) created and managed by a federated database administrator(s). In a loosely-coupled federation it is the user's responsibility to create and maintain the federation's integration regime. They may access data either by means of views defined over multiple databases [18, 19] or by defining a query using a multidatabase language [23, 24] which enables users to link concepts within queries.

With the growth and widespread popularity of the Internet, the number of databases available for public access is ever-increasing both in number and size, while at the same time the advances in remote resource access interface technologies (e.g. JDBC, ODBC) offer the possibility of accessing these databases from around the world across heterogeneous hardware and software platforms. It is envisaged that the primary issue in the future will not be how to efficiently process data that is known to be relevant, but rather to determine which data is relevant for a given user requirement or an application and to link relevant information together in a changing and evolving situation[27, 35, 36]. The creation of an environment that permits the controlled sharing and exchange of information among multiple heterogeneous databases is also identified as a key issue in future database research [17, 40]. In this environment we believe that the loosely-coupled architecture provides a more viable solution to global users' evolving and changing information requirements across the disparate databases available over a network, as it naturally supports multiple views which are dynamic, and it also avoids the need to create a global integrated schema.

In our research the main focus is on loosely-coupled federations where views are created and used to define a user's requirements to access and link together data from multiple databases. In a loosely-coupled federation, databases may join and leave as they please. In addition, global users may create, modify and enhance views in many different ways to meet their changing information requirements in a flexible manner. Some of the major issues to be addressed when

creating multidatabase views in such an environment are how to locate appropriate databases and data for some application and how to determine objects in different databases that are semantically related, and then resolve the schematic differences among these objects [2,21,29]. This must be achieved in a way that makes it possible for the federation to evolve, and easy for a user to create new views and change existing views. A wealth of research has been done on database integration but most of this work assumes that the relevant databases and data can be identified by some means during a pre-integration phase [1] and that integration is a static outcome which is comprehensive and will be relevant to all future applications. Furthermore, much published research in this area concentrates either on a one-shot integration approach which creates a single global schema or an integration via a multidatabase query [34]. Significantly, little or no emphasis has been placed on the re-usability of knowledge elicited during the schema integration process, to create different views which dynamically reflect a user's changing information requirements.

In this paper we investigate how to establish a knowledge base (KB) to assist users to create views which integrate schemas in a loosely-coupled federation of heterogeneous databases in a dynamic and flexible manner. In our approach we assume that knowledge about a database is elicited at the time it joins the federation, but is not necessarily used at that time to create an integrated schema. The knowledge base we establish evolves as new databases join the federation. It will also evolve as new views are created by users as it will capture the semantics used in creating these views. The knowledge gathered in each step is utilised in subsequent integration efforts to improve the new views created by users. The KB we propose is not a global schema. Its intention is not to create a view(s) specifically oriented towards a particular user group(s) but rather to capture information which can be used to support an environment in which users can create multiple, dynamic and flexible integrated views with less effort. Thus the knowledge base cannot be used directly as an integrated schema which helps users access data from the component databases in a federation.

We use a bottom-up strategy to build the knowledge base. With the help of DB owners, we initially use the schemas of the DBs in the federation to create a set of acyclic graphs of terms (referred to as *schema structures*). These schema structures are then merged successively with the assistance of the database owners to build the federated knowledge base. At this stage we are attempting to capture the knowledge about the databases known to their owners. Finally, the KB is utilised as a semantic dictionary to assist users in generating integrated views to access data from multiple databases in the federation when they are creating applications based on new data requirements and to find out about the information held in the federation.

For our approach, it is not a pre-requisite for the schemas of the component databases to be available in some conceptual data model such as ER or OMT. Neither do we expect to link schema components with an existing global knowledge base such as a thesaurus, WordNet [26], some form of concept hierarchies, ontologies [15,16], etc, to assign interpretations as in [2,14,25,43]. These ap-

proaches to providing a knowledge base assume that an available thesaurus will adequately cover all concepts in the federated system. Whereas we build our thesaurus from the concepts in the federation thereby ensuring its relevance to the contents of the federation.

The remainder of this paper is organised as follows: in section 2 we give a brief review of related work. An architecture to support the establishment of our knowledge base and to exploit its functionality is described in section 3. Section 4 explains the structure of the knowledge base and how it is created and modified. In section 5 we explain the functionality that the knowledge base can provide to assist users in building dynamic and flexible views. Finally, section 6 draws conclusions.

2 Related Research

In early integration methodologies [28, 30], identification of relationships between different database schema objects was considered to be the responsibility of database integrators. These integrators usually reasoned about the meaning and possible semantic relationships among schema elements in terms of their attributes and their properties (e.g. names, domains, integrity constraints, security constraints, etc). A measure based on the number of common attributes is normally used, as a basis to determine the resemblance of schema elements. The discovered relationships are then represented as assertions to be used by the schema integration tools [7, 33, 38, 41] to integrate schemas (possibly automatically). With the realisation that the manual identification of semantic relationships among schema objects is difficult, complex and time consuming and that the process cannot be fully automated [37], research began to explore the possibility of developing tools to assist users in this task.

The algorithms used by such tools to determine semantic relationships are often heuristic in nature. The fundamental operation during the identification process is comparison but the type and depth of information used for comparisons vary from tool to tool. Many of the early tools [10, 41] used techniques based on the early manual approaches and relied heavily on the information available in the schemas. They basically compared the syntactic properties of attributes in pairs to determine their equivalence, and used attribute equivalence as a heuristic to determine the semantic similarities among schema objects (e.g. relations, classes, etc). No attempt was made in this process to reuse the knowledge elicited from integrating one group of schemas in any subsequent integrations of other (related) schema groups. With the realization that schema information contents alone is not sufficient to determine the semantic similarity between schema objects [11, 21, 2], attempts were made to enrich the semantics of the schema elements prior to the integration process. This semantic enrichment is done basically in two ways: by eliciting knowledge from the database owner and building a semantic model over the schema, or by linking schema elements with existing knowledge bases (KBs). Tools that attempt to create conceptual models (e.g. ER, OMT) on top of schemas prior to integration can be consid-

ered to belong to the first category, while schema integration tools that utilise already existing ontologies, terminological networks such as WordNet, concept hierarchies, etc. fit into the second category.

Tools and techniques to support the establishment of KBs for database schema integration are reported in [2, 4, 9]. Many of the KBs proposed [2, 4] are strict concept hierarchies where the generalisation/specialisation relationship is used to organise concepts. Other types of relationship such as positive-association and aggregation are included in a number of projects [13, 43], but how these types of relationship are incorporated among the concepts in the KB is not mentioned. To the best of our knowledge the inference of semantic relationships among schema components (or DBs) by pooling the knowledge about several databases into a common repository which is then used as the KB to support the integration process is novel.

3 System Architecture

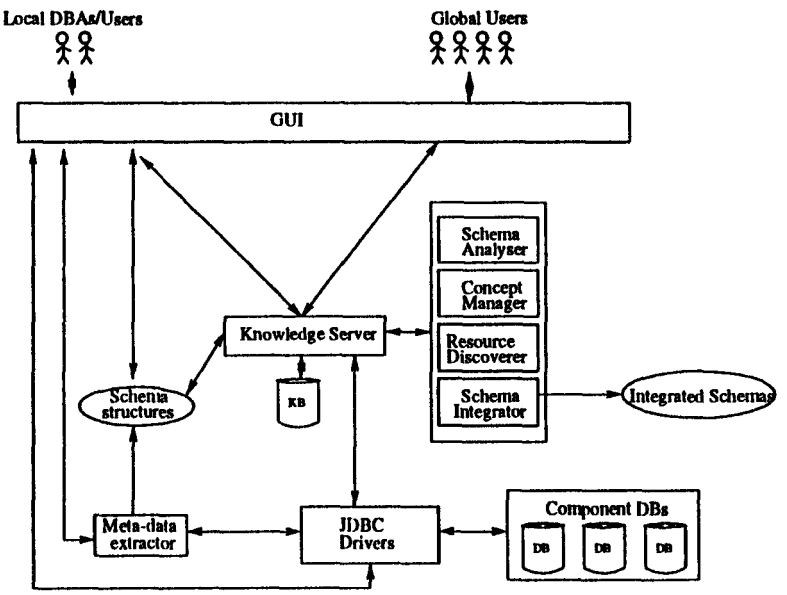


Fig. 1. The Architecture of our System

The architecture we adopt is depicted in Figure 1. Its major components are the Meta-Data Extractor (MDE), the Knowledge Server (KS) and the Graphical User Interface (GUI). The MDE interacts with the component databases (DBs) in the federation via JDBC drivers to extract their intensional data, and builds an associative network based model (*schema structure*) locally for each participating

DB in the federation. These schema structures are then analysed to discover implicit relationships among different schema elements, enriched with semantics elicited from the DB owners, and merged with the KB. The KB is maintained by the KS which provides the functionality necessary for its establishment and evolution and for its role in assisting users in their integration efforts.

The GUI component is the primary interface between the users, the knowledge server and the component databases. It allows the DB owners to view the schema structures created by the MDE as acyclic graphs of terms and provides functionality to modify schema structures and enhance the semantics of the elements in them. In addition the global users may use the GUI to invoke functionality provided by the KS.

3.1 Schema structures

```
catalogued_book(isbn,author,title,publisher,year,classmark)
reservation(isbn,name,address,date_reserved).
book_loan(book_no,isbn,shelf,name,address,date_due_back)
person(name,address,status)
loan_status(status,loan_period,max_num_books).
```

Fig. 2. An example of a specimen relational schema (based on a schema from [31])

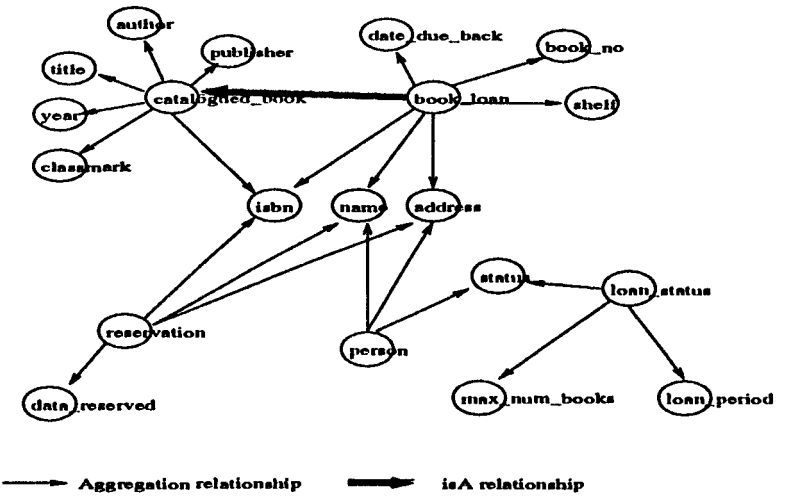


Fig. 3. An example of a Schema structure

The primary information source used to build (and evolve) the KB is the meta-data of the component databases in a federation. The DBs in the federa-

tion may be implemented in different database management systems (DBMSs) with different data models (e.g. relational, hierarchical, network, object oriented, etc.). This type of heterogeneity in a multidatabase system is referred to as *syntactic heterogeneity* [5, 3]. The solution commonly adopted to overcome syntactic heterogeneity is to use a canonical data model and to map all schemas in a federation to this canonical model. We have determined that the meta-data required to build the KB can be provided by building associative networks over each database in the federation. Hence in our approach the canonical data model used is based on associative networks, and the meta-data models generated on top of the component DBs by using this data model are referred to as *schema structures.*

3.2 Canonical data model

We organise the meta-data of a schema as an associative network of terms, where terms represent either schema objects (e.g. relation, entity, class, etc.) or attributes of schema objects defined in the corresponding DB schema. Terms are related with other terms via binary relationships (*links*). The terms and links have *properties* some of which are essential. Each property is given a *label* (e.g. name, strength, etc.) and has a *value(s).* The labels classify properties into *types.* In the following sections we use labels to identify properties. Two essential properties of a link are its *strength* and its *link-type.* The strength of a link represents how closely the connected terms are related in the DB and takes a value in the interval (0, 1]. A link with strength 1 indicates a definite relationship. All the other values represent tentative associations. The link property *link-type* can take a value from the set {aggregation, isA, positive-association, synonym}.

The links between terms in a schema structure are generated as follows :

aggregation : links two terms T_1 and T_2 when T_1 corresponds to a schema object O_1 and T_2 corresponds to an attribute A_i of the object O_1.

isA : links terms T_1 and T_2 when T_1 and T_2 denote two schema objects O_1 and O_2, respectively, and a specialisation/generalisation relationship exists between them.

positive-association : links terms when an association other than *aggregation* and *isA* is defined between objects in the schema.

synonym : links two differently named schema terms when they correspond to two schema object attributes that denote the same real-world aspect. For example in relational DBs, the same attribute may exist with different names in different relations to create explicit relationships between relations.

Since *aggregation*, *positive-association* and *isA* links are used to link terms only when such associations are explicitly defined or can be identified by the DB owners, their strengths are always assigned the value 1. The tentative associations may be generated among terms by using schema analysis tools to guide

DB owners to discover implicit relationships among schema elements. Once such relationships are discovered, DB owners may transform tentative links to definite links by changing their strength to 1. The implementation characteristics of schema objects and their attributes (e.g. data type, scale, precision) may be coded in schema structures as properties of their corresponding terms.

In Figure 2 a specimen relational schema is shown and its corresponding generated schema structure is shown in Figure 3. We expect the DB owners to identify different schema object attributes in their schema structures which denote the same real-world aspects and to merge such attributes as a single term (e.g. isbn in Figure 3) or to relate them with *synonym* links before merging the schema structures with the KB.

4 Structure, Creation and Modification of the KB

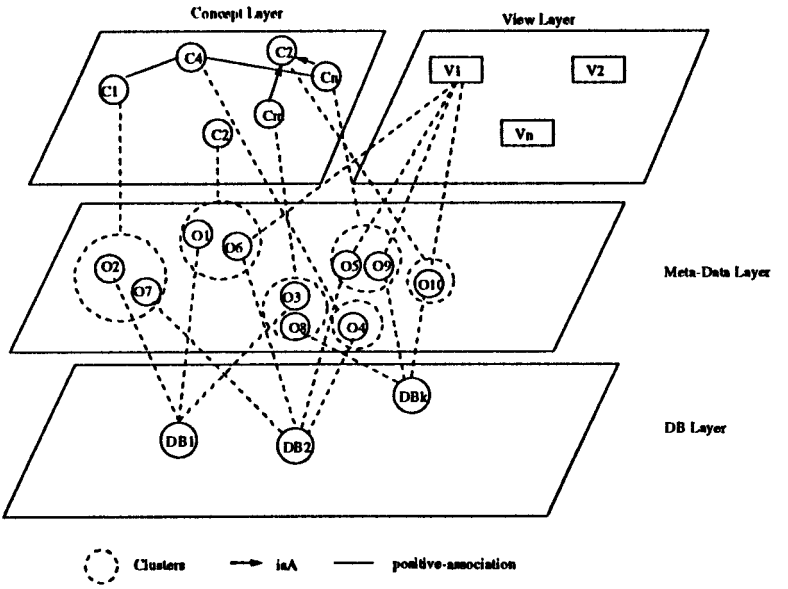


Fig. 4. The Structure of the KB

The KB can be viewed as composed of four layers: a *concept layer*, a *view layer*, a *meta-data layer* and a *DB layer*, as shown in Figure 4. The top left layer comprises concepts, each of which represents an object cluster in the meta-data layer. A concept in the top left layer can be thought of as a unit of knowledge representation that guides the process of classifying schema objects into clusters, and a cluster can be considered as a logical grouping of schema objects. Concepts may be related with each other via binary relations (*links*). As in schema

structures, links have properties. In addition to *strength* and *link-type* properties, they also have an essential property named *frequency*, which specifies how many times the corresponding link occurred in the federation and is being used in user views. In the concept layer, link strengths represent how closely concepts are related in the federation. The link property *link-type* here takes values from the set {aggregation, isA, positive-association}.

In the concept layer, *positive-associations* represent possible associations among concepts, and are generated as described in the following sections. Concepts possess attributes providing descriptive information about themselves. The attributes are connected with their corresponding concepts through aggregation links. When an attribute of a concept is further described by using a set of attributes, it is treated as another concept. Hence a concept may have attributes which are also concepts. Both concepts and attributes may possess properties. Two essential properties of concepts and attributes are *name* and *frequency*. A concept is allowed to have multiple names in the concept layer. The frequency of a concept (or a concept attribute) specifies in how many databases in a federation the concept (or the attribute) is defined.

The view layer stores information relevant to views generated by users over the federation. Its main function is to gather knowledge required to determine and notify view users, DB meta data changes that potentially affect their views.

The meta-data layer maintains all the schema structures submitted to the KS, by organising all terms that represent objects in schema structures into clusters (*object clusters*). All corresponding objects in a cluster are hypothesised to be semantically similar. This layer is intended to be used to assist users in reconciling schematic differences among semantically similar items during schema integration.

The DB layer maintains information about the databases whose schema objects are represented in the meta-data layer. It is used to link to the extensional data.

4.1 Merging schema structures with the KB

The KB is created by merging, in ladder fashion [1], the schema structures of the component DBs. Merging a schema structure with the KB changes all four layers of the KB. At the DB layer a new node is created to capture the information of the DB being merged. All meta-data in the schema structure is added to the meta-data layer and the terms in the schema structure are merged with concepts in the concept layer as described below.

4.2 Merging schema structures with the concept layer

During this process some of the terms in schema structures are merged with concepts in the concept layer while others are merged with attributes of existing concepts. The approach we use to merge a schema structure with the KB comprises five stages: *common term set generation, sub-net(s) generation, sub-net(s) and schema structure unification, merge terms with concepts*, and *merge links*,

which are carried out once in that order to merge a schema structure with the KB. The first three stages of this process are aimed at establishing mappings between schema structure terms and concepts by considering all schema terms as a single unit of related terms. The last two stages assimilate schema structure knowledge into the KB.

In the *common term set generation* stage, a set S of terms common to both the schema structure and the KB is generated. The common terms may exist in the KB either as concepts or as attributes of concepts.

During the *sub-net(s) generation* stage, terms in the set S are spanned along aggregation and positive-association links in the concept layer whose strengths exceed a given threshold value, to obtain a sub-net (or a set of disjoint sub-nets) of concepts that span all terms in S. We use the following algorithm to generate the sub-net(s) that span all the common terms in the set S. Initially a term t_0 in the set S is selected as a seed term, and all concepts in the concept layer with the same name are taken as starting concepts. All starting concepts are then spanned to a given depth (d), generating a number of distinct sub-net(s) of concepts. We define the span depth as the number of links traversed by the spanning process from the starting concept. Out of these sub-nets, the sub-net having the maximum number of terms in the set S is selected as the current sub-net. In the next step, a new set S is created by removing the terms that are in the current sub-net from the initial set S. One of the terms in this new set is selected as the new seed and its corresponding concepts are spanned as above, until either one of the concepts generated during the spanning process matches a concept in the current sub-net or all concepts span to the maximum depth, d, without matching a concept in the current sub-net. In the former case, the sub-net generated by the spanning process is merged with the current sub-net to generate a new current sub-net to be used in the next iteration. In the latter case, the sub-net(s) that has the maximum number of terms in the set S is selected as the best span and is added to the current sub-net. Then a new set is created by removing the terms in S that appear in the best span, and the process is repeated till the set S is empty.

In the *sub-net(s) and schema structure unification* stage, concepts in the spanned sub-net(s) are unified with terms in concept structures that represent schema objects to determine the most suitable concept for each term to merge with. The final result of this stage is a set of ordered pairs $(T_i,\{C_1,C_2,C_3...C_n\})$, where T_i is a term in the schema structure representing an object and the C_is represent a suitable concept for T_i to merge with. With the assistance of the DB owners, the above generated ordered pairs are converted into a set of ordered pairs of the form (T_i,C_i), where C_i is the concept with which the term T_i is to be merged in the next stage. Then the T_is in the meta-data layer are moved into the clusters represented by the concept C_is. When several concepts match closely to a given schema object term, then all such matching concepts that are not linked via a common parent are also unified to determine their inter-relationships and the results are shown to the DB owner. He may inspect these results to discover

scattered descriptions of the same concepts. If such concepts are found, they may be merged together.

In the *merge terms with concepts* stage, terms in the schema structure are merged with the identified concepts in the concept layer and the properties (e.g. frequency) of the concepts and their attributes are updated appropriately. For the schema object terms where no suitable terms are found in the concept layer, new concepts are created.

During the *merge link stage*, new links may be created among concepts, and the existing links properties are updated. We create a positive-association link (or update their frequencies) among two concepts C_i and C_j when either a positive association is defined between T_i and T_j in the schema structure or there is a path over the aggregation and positive-association links whose length is less than a given value between T_i and T_j in the schema structure. Based on the frequency values of concepts and links, a number of different computations can be devised to calculate link strengths. After experimenting with a number of different computations in our prototype, we chose to use the following equations to compute link strengths, by taking into account the similarity measures established in cluster analysis [12]. We have observed that the selected computations yield a fewer number of more focused disjoint sub-nets during sub-net generation stage.

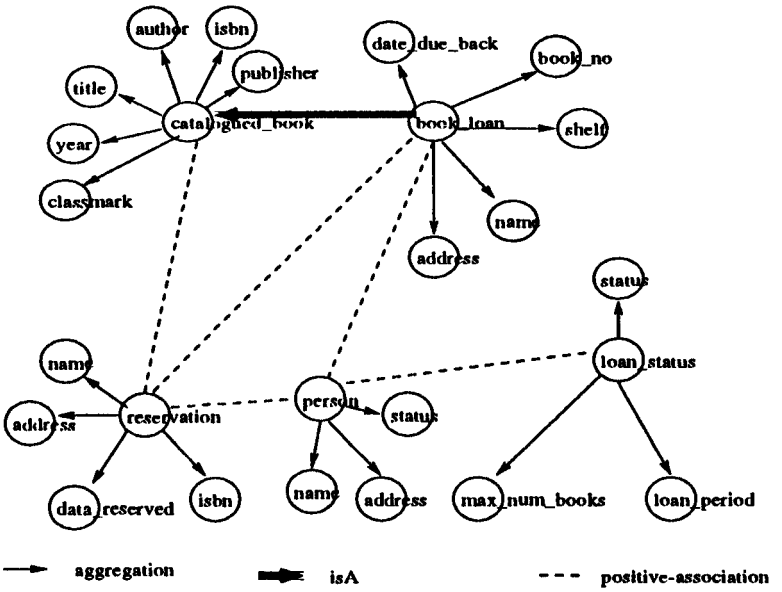


Fig. 5. Structure of KB concept layer after merging the schema structure shown in figure 3 with an empty KB

– aggregation link strength connecting concept p with its attribute i = F^a_{pi}/F^c_p

- positive-association link strength connecting the concept i with the concept j = $F^l_{ij}/\min(F^c_i,F^c_j)$
- isA link strength = 1

where F^c_i is the frequency of concept i, F^c_{pi} is the frequency of attribute i of concept p, F^l_{ij} is the frequency of the link connecting concepts i and j.

The resulting conceptual layer of the KB is shown in Figure 5, after the schema structure shown in Figure 3 is merged with an initial empty KB.

5 Functionality of the KB

In the KB, schema objects considered to be semantically similar are grouped into clusters and a concept is attached at a higher level to represent each cluster. In addition, links are maintained between schema objects and databases from which they came. Hence this knowledge organisation allows users to start discovering databases relevant for some application (DB discovery) from two levels, from the concept layer or from the DB layer. A user can initiate the DB discovery process from the concept layer by specifying a set of concepts $\{C_1,C_2,C_3...C_n\}$ that he thinks might be relevant for his application. These concepts, C_is, lead to a set of clusters, and then to a collection of schema objects $\{O_1,O_2,O_3...O_m\}$ at the meta-data layer. By moving down from these schema objects, O_is, along the links maintained between the meta-data layer and DB layer, a set of databases $\{DB_1,DB_2,...DB_k\}$ in which the schema objects are defined can be determined. The set $\{DB_1,DB_2,...DB_k\}$ is then sorted on a weight computed by considering the concepts to which these DB_is, relate and are presented to the user. Users may alternatively start the DB discovery process from the DB layer by selecting a candidate DB(s) from the federation. In this case the selected DB(s) is used to generate a set of concepts at the concept layer by moving along the links between the DB layer and the meta-data layer, followed by the links between the meta-data layer and the concept layer. The generated set of concepts are then used as before to discover DBs in the federation that define relevant schema objects.

Once a set of DBs relevant for some application is chosen, users may determine the relationships among the schema objects in them by determining their clusters, the concepts representing those clusters, and the associations existing among clusters at the concept layer. The schema objects belonging to the same cluster are considered to represent semantically similar real world entities, while the semantic relationship of schema objects belonging to different clusters can be determined by considering the relationships between the concepts represented by these clusters.

When views are created over DBs in the federation, users may integrate schema objects in many different ways. The existing links used and the new links generated during these integration efforts are also recorded at the concept layer and this information is utilised to assist users in subsequent integrations.

The links maintained between each layer facilitate the easy modification of information in the three layers of the KB, during DB schema modification. When

a DB leaves the federation the KB objects corresponding to it at the DB layer and meta-data layer are removed and the frequencies of relevant concepts and links at the concept layer are updated to reflect this change.

6 Conclusion and future work

We have described how a KB can be established over a loosely-coupled federation of heterogeneous DBs to assist users to integrate schemas. Our research is guided by the assumptions that:

- The schemas developed to represent similar real world domains typically share a set of common terms as schema object names and their attribute names.
- Semantic relationships among schema elements in a federation can be identified better by pooling meta-data of component DBs.

We do not depend on any pre-existing global taxonomies/dictionaries in building the KB. The KB we establish is dynamic and evolves with the federation. It is more germane to the federation as it uses mainly the meta information of the DBs from the schemas and the DB owners, and more comprehensive as it stores information necessary for a wide variety of activities such as information browsing, planning/preparing for an integration, resolving schematic conflicts, accessing data, etc.

An approach based on generating a sub-net spanning a set of terms, followed by unification, is used to merge DB meta-data with the KB. How the knowledge generated by pooling meta-data of component DBs is used to determine the strength of associations of concepts within the federation, and how these strengths are utilised to enhance the semantics of schema elements has been explained. Finally we described how the organisation of the KB helps users in database discovery and in determining semantic relationships among schema objects.

The architecture and algorithms we have presented in this paper are based on a prototype system built to support users creating views over a set of library databases. This prototype was built to demonstrate that our knowledge base could be established and utilised to create views. Two library databases have been linked in the federation and a number of views have been created utilising the resulting knowledge base to assist in view creation. We intend to extend our research by building a work-bench of tools to assist users to establish the KB and to utilise its functionality in their integration efforts. As our KB stores information at the DB layer to retrieve data from the component databases in the federation, we are also aiming to develop a query processor to materialise user views created over the federation. In addition we intend to explore the possibility of using the global KB such as WordNet, ontologies, etc. to enrich and verify our KB.

References

1. C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986.
2. M. W. Bright and A. R. Hurson. Linguistic support for semantic identification and interpretation in multidatabases. In *Proceedings of the First International Workshop on Interoperability in Multidatabases*, pages 306–313, Los Alamitos, California, 1991.
3. M. W. Bright, A. R. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3), March 1992.
4. S. Castano and V. De Antonellis. Semantic dictionary design for database interoperability. In *Proceedings of the 13th International Conference on Data Engineering*, pages 43–54, Birmingham, U.K., April 1997.
5. M. Castellanos and F. Saltor. Semantic enrichment of database schemas: An object oriented approach. In Y. Kabayashi, M. Rusinkiewiccz, and A. Sheth, editors, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 71–78, Kyoto, Japan, 1991.
6. A. Chatterjee and A. Segev. Data manipulation in heterogeneous databases. *SIGMOD RECORD*, 20(4), December 1991.
7. S. Chawathe, H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinous, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 10th IPSJ Conference*, pages 7–18, Tokyo, Japan, October 1994.
8. C. Collet, M. N. Huhns, and W. Shen. Resource integration using a large knowledge base in Carnot. *IEEE Computer*, pages 55–62, December 1991.
9. S. Dao, D. M. Keirsey, R. Williamson, S. Goldman, and C. P. Dolan. Smart data dictionary: A knowledge-object-oriented approach for interoperability of heterogeneous information management systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 88–91, Kyoto, Japan, 1991.
10. U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, SE-10(6):628–645, November 1984.
11. B. EagleStone and N. Masood. Schema interpretation: An aid to schema analysis in federated database design. In *Engineering Federated Database Systems (EFDBS97), Proceedings of the International CAiSE97 Workshop*, University of Magdeberg, Germany, June 1997.
12. B. S. Everitt. *Cluster Analysis*. Edward Arbold, 3rd edition, 1993.
13. P. Fankhauser and E.J. Neuhold. Knowledge-based integration of heterogeneous databases. In *Proceedings of the IFIP WG 2.6 Conference on Semantics of Interoperable Database Systems(DS-5)*, pages 155–175, Lorne, Victoria, Australia, November 1992.
14. A. Goni, E. Mena, and I. Illarramendi. Querying heterogeneous and distributed data repositories using ontologies. In *Proceedings of the 7th European-Japanese Conference on Information Modelling and Knowledge Bases (IMKB'97)*, Toulouse, France, May 1997.
15. T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199-220, 1993.

16. T.R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University, 1993.
17. J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *International Journal of Intelligent & Cooperative Information Systems*, 2(1):51–83, 1993.
18. D. Heimbigner and D. McLeod. Federated information bases - a preliminary report. In *Infotech State of the Art Report, Databases*, pages 223–232. Pergamon Infotech Limited, Maidenhead, Berkshire, England, 1981.
19. D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985.
20. V. Kashyap and A. Sheth. Schema correspondences between objects with semantic proximity. Technical Report DCS-TR-301, Department of Computer Science, Rutgers University, October 1993.
21. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: a context-based approach. *VLDB Journal*, 5:276–304, 1996.
22. W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, pages 12–18, December 1991.
23. W. Litwin. From database systems to multidatabase systems: Why and how. In *Proceedings of the 6th British National Conference on Databases(BNCOD6)*, pages 161–188, Cardiff, U.K., 1988.
24. W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.
25. E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems(CoopIS'96)*, Brussels, Belgium, June 1996.
26. G. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11), November 1995.
27. S. Milliner, A. Bouguettaya, and M. Papazoglou. A scalable architecture for autonomous heterogeneous database interactions. In *Proceedings of the 21st VLDB Conference*, pages 515–526, Zurich, Switzerland, 1995.
28. A. Motro and P. Buneman. Constructing superviews. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 56–64, 1981.
29. S. Navathe and A. Savasere. A practical schema integration facility using an object-oriented data model. In O. A. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems, A Solution for Advanced Applications*. Prentice Hall, 1996.
30. S. B. Navathe and S. G. Gadgil. A methodology for view integration in logical database design. In *Proceedings of the 8th VLDB Conference*, pages 142–164, Mexico City, Mexico, September 1982.
31. Elizabeth Oxborrow. *Databases and Database Systems, Concepts and Issues*. Chartwell-Bratt(Publishing and Training Ltd), Sweden, 2nd edition, 1989.
32. E. Pitoura, O. Bukhres, and A. K. Elmagarmid. Object-oriented multidatabase systems: An overview. In O. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems, A Solution for Advanced Applications*. Prentice Hall, 1996.
33. M. A. Qutaishat. *A Schema Meta Integration System for a Logically Heterogeneous Object-Oriented Database Environment*. PhD thesis, Department of Computing Mathematics, University of Wales College of Cardiff, Cardiff, 1993.

34. A. Rosenthal and L. J. Seligman. Data integration in the large: The challenge of reuse. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, September 1994.
35. A. Sheth and V. Kashyap. So far (Schematically) yet so near (Semantically). In *Proceedings of TC2-IFIP WG 2.6 Conference on Semantics of Interoperable Database Systems (DS-5)*, pages 283–312, Lorne, Victoria, Australia, November 1992.
36. A. P. Sheth. Semantic issues in multidatabase systems. *SIGMOD RECORD*, 20(4):5–9, December 1991.
37. A. P. Sheth, S. K. Gala, and S. B. Navathe. On automatic reasoning for schema integration. *International Journal of Intelligent and Co-operative Information Systems*, 2(1):23–50, March 1993.
38. A. P. Sheth, J. Larson, A. Cornelio, and S. B. Navathe. A tool for integrating conceptual schemas and user views. In *Proceedings of the 4th International Conference on Data Engineering*, pages 176–182, Los Angeles, CA, February 1988.
39. A. P. Sheth and J. P. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990.
40. A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *SIGMOD RECORD*, 19(4):23–31, December 1990.
41. J. De. Souza. SIS: A schema integration system. In *Proceedings of the 5th British National Conference on Databases (BNCOD5)*, pages 167–185, Canterbury, U.K., July 1986.
42. S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1:81–126, July 1992.
43. C. Yu, W. Sun, S. Dao, and D. Keirsey. Determining relationships among attributes for interoperability of multi-database systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 251–257, Kyoto, Japan, April 1991.

Considering Integrity Constraints During Federated Database Design *

Stefan Conrad Ingo Schmitt Can Türker

Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D-39016 Magdeburg, Germany

{conrad|schmitt|tuerker}@iti.cs.uni-magdeburg.de

Abstract. Correct transformations and integrations of schemata within the process of federated database design have to encompass existing local integrity constraints. Most of the proposed methods for schema transformation and integration do not sufficiently consider explicit integrity constraints. In this paper we present an approach to deal with integrity constraints. Our approach bases on the idea to relate integrity constraints to extensions. A set of elementary operations for schema restructuring and integration is identified. For these operations we define major rules for dealing with integrity constraints. By means of a small example we then demonstrate the application of these rules.

1 Introduction

In many large organizations, different legacy data management systems are maintained and operated. These data management systems and the data managed by them have developed independently from each other. Among these systems there are database management systems but also merely file-based systems differing in several aspects such as data model, query language, and system architecture as well as the structure and semantics of the data managed. In this context the term 'heterogeneity of the data management systems' is commonly used. As a rule, applications for specific needs continue to be based on such systems. More recent applications often require access to the distributed databases but their implementation fails due to heterogeneity. *Federated database systems* [SL90] are designed to overcome this problem by integrating data of legacy database systems only virtually. That is, 'only' the local schemata are integrated to a federated schema. Global applications can retrieve and update federated data through the federated schema.

The main task during the design of a federated database is the transformation and integration of existing local schemata into a federated schema. The transformation [EJ95] and integration [LNE89,SPD92,RPG95,SS96a] of local schemata into a federated schema must fulfill the following requirements:

1. For each query against a local schema there exists a global query against the federated schema returning the same query result. This requirement avoids loss of information during the federation.

* This work was partly supported by the German Federal State Sachsen-Anhalt under grant number FKZ 1987A/0025 and 1987/2527R.

2. Each global update or insert operation can be propagated to the local DAMS correctly. That is, the local schemata must not be more restrictive than the federated schema. Therefore, schema transformation and integration have to encompass integrity constraints. Global integrity constraints can be used to optimize global queries and to validate global updates. In addition to the transformed local integrity constraints the federated schema contains 'integration constraints' [BC86] stemming from inter-schema correspondence assertions [SPD92]. These integration constraints force global consistency.

The second requirement demands to transform integrity constraints during the federation design process. In our approach, we consider integrity constraints referring to class extensions. A typical schema conflict occurs if extensions of two local classes can overlap. This overlapping itself is an integration constraint. By adding such integration constraints, local integrity constraints can come into conflict. That is, for objects simultaneously stored in both classes different conflicting integrity constraints can be defined. The problem in this case is to find a federated schema fulfilling the requirements mentioned above. Various publications tackle this conflict. [BC86,Con86], for instance, describe the integration of views and name local schemata (views) without conflicting integrity constraints conflictfree schemata. In general, testing for conflictfreeness is undecidable. In those approaches only conflictfree schemata (views) can be integrated correctly. [RPG95,EJ95] show how to integrate schemata into a federated schema considering integrity constraints. Conflicting integrity constraints are not integrated. [VA96] distinguish between subjective and objective constraints. Subjective constraints are only valid in the context of a local database and need not to be dealt with during schema integration. In this way only objective and non-conflicting integrity constraints are integrated.

In [AQFG95] the integration of conflicting integrity constraints bases on a conflict classification. In some cases, the more relaxed constraint is taken over to the integrated class. In other cases, the local constraints are reformulated in an implication format and attached to the integrated class.

In contrast to the mentioned approaches we integrate integrity constraints basing on set operations on class extensions. Extensions of local classes have to be decomposed into disjoint base extensions and later to be composed to the federated schema (cf. [SS96a,SC97,CHS+97]). Local integrity constraints are related to class extensions. This idea is already mentioned shortly in [BCW91]. By decomposing and composing class extensions in a well-defined way, we are able to derive integrity constraints for integrated schemata (as well as for arbitrary views, e.g. external schemata) in a clean and logical sound way. We tackle the problem of conflictfreeness of local schemata in a different way than most other approaches mentioned before. It is possible to discover conflicting integrity constraints during database schema integration. Such conflicts are often a hint that there are other conflicts behind that, e.g. a semantic conflict or a name conflict which has not been discovered up to that moment.

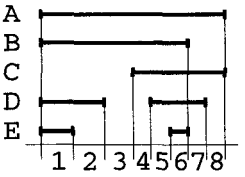
In this paper we focus on the integration of integrity constraints based on the integration of the extensions (classes) they belong to. Thereby, we continue the work presented in [CHS+97] where a basic analysis of the problem of integrating integrity constraints was given. The next section introduces elementary operations for decomposing and composing class extension. For these operations we present rules to transform and integrate integrity constraints. We briefly dis-

cuss the application of these rules to classes of frequently occurring integrity constraints before we give a detailed example.

The rules presented in this paper are not intended to be applied in a fully automatic manner. The designer of the federated database has to control the application of them. In this way he can see during the integration process whether there are conflicting constraints and in which way the integration has to be changed for resolving such conflicts.

2 Elementary Operations

During the schema integration process the main task is to find all relationships (integration constraints) between the class extensions in the base schemata which are to be integrated. Two class extensions can be always equal or always disjoint. In addition to these relationships, two class extensions can overlap semantically. A special case of overlapping class extensions is that one is always a subset of another one (cf. w.r.t. these relationships for instance [SPD92]). As showed in [SS96b], pairwise comparisons of class extensions is not always sufficient because not all pieces of information necessary for an adequate integration can be found in this way. Instead, all class extensions must be considered at the same time. The result of such an analysis can be presented in a graphical manner as depicted as follows:



The horizontal lines represent the class extensions of five classes A, B, C, D, and E, respectively. For instance, the class extensions of B and C are overlapping. Furthermore, E is a subclass of D. Based on this information, a set of *base extensions* can be identified as a canonical representation. In the figure, eight base extensions have been found and numbered by 1 to 8.

Using the standard set operations *intersection* and *difference* we can describe the base extensions in terms of the original class extensions. After such a decomposition into base extensions we can construct each possible class extension which might occur in the integrated schema by applying the standard *union* operation to a collection of base extensions. Based on this simple procedure of decomposing and then composing class extensions, resolving of extensional conflicts can be described in terms of the three standard set operations. In [SS96a] an algorithm is proposed to compose base extensions to a federated schema automatically. Due to the fact that *selection* is a very important operation for defining views or specifying export schemata, we also consider *selection* to be an elementary operation.

The decomposition of class extensions can produce base extensions which belong to local classes from different databases. Objects of such base extensions are managed by more than one database system simultaneously. The values of common attributes are stored redundantly. If the local databases store the current state of real-world objects correctly then the values of the common at-

tributes would be equal. Otherwise we can state that different values indicate inconsistency.

For two local classes of which the extensions overlap conflicting integrity constraints concerning common attributes can exist. For instance, the attribute salary can be restricted by 'salary > 3000' in one database and by 'salary > 2500' in another database. Such conflicting integrity constraints defined for common attributes of objects stored in both databases are indications of

1. wrong specification of extensional overlappings or
2. wrong specification of attribute-attribute relationships (attribute conflicts) or
3. incomplete or incorrect design of at least one local schema.

A wrong analysis of the schemata to be integrated (first and second item) can be corrected without violation of local autonomy. A wrong design of a local schema is more tricky. If both databases are consistent, i.e. the values of common attributes are equal, then the integrity constraints of both classes hold simultaneously. In this case the local integrity constraints can be corrected without invalidating the existing database. The corrected local integrity constraints would then help to detect wrong local updates.

However, sometimes the local database should continue in possibly inconsistent states due to the demand for design autonomy. A solution then is to regard common attributes as semantically different. In this way, the integrity constraints are not longer in conflict.

The next section presents rules for treating integrity constraints during the decomposition and composition of class extensions.

3 Rules for Dealing with Integrity Constraints

For presenting the rules which can be applied to integrity constraints during schema integration we first need to introduce our notation and to define some important terms.

As already mentioned before, integrity constraints are usually given for a certain set of objects. Therefore, we always consider constraints together with the sets of objects for which they are valid. We denote by $\Theta_\phi(E)$ that the constraint ϕ holds for a set E of objects. Typically, E is a class extension. From a logical point of view, $\Theta_\phi(E)$ says that all free variables occurring in ϕ are bound to the range E. The range E is also called the *validity range* of the constraint ϕ.

For simplifying the presentation we distinguish between two basic kinds of constraints. An *object constraint* is a constraint which can be checked independently for each object in the validity range of that constraint. Integrity checking for object constraints is easy and can be done very efficiently. In case we insert a new object or modify an existing object, we only need to consider that object for checking object constraints.

In contrast, a *class constraint* can be characterized as a constraint which can only be checked by considering at least two objects out of its validity range. Often, all currently existing objects in the validity range are needed for checking a class constraint.[1] Obviously, it is possible to distinguish several different kinds

[1] The distinction between object constraints and class constraints follows the taxonomy of integrity constraints presented in [AQFG95].

of class constraints, but, for the purposes of this paper, we do not need such a distinction.

In the following, we consider the four elementary operations identified in the previous section. For each elementary operation we give a rule for dealing with object constraints. Afterwards, we briefly discuss in which cases these rules can also be applied to class constraints.

Selection. Applying a selection σ_ψ (where ψ is the selection condition) to an extension E results in a subset of E denoted by $\sigma_\psi(E)$. The selection condition ψ can be considered as an object constraint. In consequence, ψ holds for the extension $\sigma_\psi(E)$. Assuming that an object constraint ϕ holds for E, we can conclude that $\phi \wedge \psi$ holds for $\sigma_\psi(E)$:

$$\Theta_\phi(E) \rightarrow \Theta_{\phi\wedge\psi}(\sigma_\psi(E)).$$

Difference. Applying the difference operation results in a subset of the first operand. Therefore, the difference operation and taking a subset can be dealt with in the same way with respect to object constraints. If an object constraint ϕ holds for an extension E, ϕ also holds for each subset of E. This is due to the fact that ϕ is an object constraint which can be checked independently for each object in E and this is true for each subset of E as well. This property leads to the following rule (where $E = E_1 \cup E_2$ and, thus, E_1 is a subset of E):

$$\Theta_\phi(E_1 \cup E_2) \rightarrow \Theta_\phi(E_1).$$

Intersection. The intersection of two (or more) extensions is always a subset of each of the original extensions. Due to the fact that object constraints can be transferred from an extension to any subset of that extension (as explained before), the following rule holds for object constraints. If an object constraint ϕ holds for an extension E_1 and an object constraint ψ holds for an extension E_2, then both ϕ and ψ (and thereby also the logical conjunction $\phi \wedge \psi$) hold for the intersection of E_1 and E_2:

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi\wedge\psi}(E_1 \cap E_2).$$

Union. In case we unite two extensions E_1 and E_2 we can only use the knowledge that each object in $E_1 \cup E_2$ is an object in E_1 or in E_2. In consequence, if ϕ is an object constraint valid for E_1 and ψ an object constraint valid for E_2, then $\phi \vee \psi$ is valid for $E_1 \cup E_2$:

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi\vee\psi}(E_1 \cup E_2).$$

In Figure 1 we have listed all rules introduced so far (rules (1) to (4)). In addition, the rules (5) and (6) express important logical properties.

In case an integrity constraint consists of a conjunction of several conditions, each of these conditions can be considered as a single integrity constraint valid for the same extension (rule (5)). Furthermore, we can weaken each constraint by disjunctively adding an arbitrary further condition (rule (6)). Rule (7) says that two constraints being valid for the same extension may be put together into

Let ϕ and ψ be object constraints, then the following rules hold:

$$\Theta_\phi(E) \rightarrow \Theta_{\phi \wedge \psi}(\sigma_\psi(E)) \quad (1)$$

$$\Theta_\phi(E_1 \cup E_2) \rightarrow \Theta_\phi(E_1) \quad (2)$$

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi \wedge \psi}(E_1 \cap E_2) \quad (3)$$

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi \vee \psi}(E_1 \cup E_2) \quad (4)$$

$$\Theta_{\psi \wedge \phi}(E) \rightarrow \Theta_\psi(E) \quad (5)$$

$$\Theta_\psi(E) \rightarrow \Theta_{\psi \vee \phi}(E) \quad (6)$$

$$\Theta_\phi(E), \Theta_\psi(E) \rightarrow \Theta_{\phi \wedge \psi}(E) \quad (7)$$

$$\rightarrow \Theta_{\mathsf{true}}(E) \quad (8)$$

Fig. 1. Overview of rules for *object constraints*

a conjunction which is still valid for that extension. For completeness, rule (8) allows to introduce the constraint true being valid for each extension.

Considering rule (7) we can see that it is only a special case of rule (3) where E_1 equals E_2. Because this is an important case often occurring during schema integration, we give an own rule for that.

In case E_1 equals E_2 we can also apply rule (4). Then, $\phi \vee \psi$ can be derived for E_1 (and E_2, resp.). From a formal point of view this result is correct, although it is too weak. In general, we are interested in obtaining the strongest constraint(s) which can be derived. Nevertheless, the rules allow us to derive weaker constraints.

Up to now, we have only dealt with object constraints. Due to the fact that the rules (5)–(7) describe general logical properties they can be used for arbitrary constraints including class constraints without modification. For the other rules we have now to see in which cases they can be applied to class constraints.

4 Applicability of the Rules to Class Constraints

The rules presented in the previous section hold for object constraints. The questions now is whether these rules are valid for class constraints, too. In this section we investigate certain kinds of class constraints (uniqueness constraints, referential integrity constraints, and constraints based on the aggregate functions max and min) which are relevant in practice.

4.1 Uniqueness Constraints

The rules (1), (2), and (3) also hold for uniqueness constraints (unique). Rule (1) means that all (selected) objects of the resulting set fulfill the uniqueness constraint besides the selection condition. Obviously, uniqueness constraints hold for each subset of a class extension for which they are originally defined. This follows immediately from the fact that an object which can be unambiguously identified in a class extension can also be unambiguously identified in each subset of this class extension. The rules (2) and (3) also hold for the same reason.

However, the following aspect has to be considered for all three rules: A uniqueness constraint which is originally defined on a superset is generally weakened by using the rules (1), (2), and (3). We will illustrate this aspect by using an example: Suppose, there is a class extension employees for which a uniqueness

constraint is defined. Further assume that this class extension can be decomposed into two base extensions project group 1 and project group 2. In this case, we can state that the uniqueness constraint holds for both base extensions. Nevertheless, the "decomposition" of the uniqueness constraint leads to a weakening because the relationship between the base extensions is not considered. Now, it is possible that in both base extensions an object can be inserted with the same key attribute values without violating one of the two derived uniqueness constraints. However, from the point of view of the original class extension the originally defined uniqueness constraint is violated.

Thus, we require a further rule which maintain the original integrity constraints. Consequently, we have to introduce the following *equivalence* rule

$$\Theta_\phi(E) \leftrightarrow \Theta_\phi(E_1 \cup E_2) \tag{9}$$

which says that a uniqueness constraint ϕ which holds for an extension E ($E \equiv E_1 \cup E_2$) also holds for the union of the both decomposed extensions E_1 and E_2 (and vice versa). The equivalence rule is not restricted to uniqueness constraints only; obviously, it holds for all kinds of constraints.

The rule (4) does not hold for uniqueness constraints on class extensions because this kind of constraints can be violated by the objects of the second class extension to be united. Thus, we cannot conclude that a uniqueness constraint holds for the united extension. This is also the case when in both class extension the same uniqueness constraint is defined (i.e. unique defined on the same attribute or attribute combination).

Let us consider the following example to illustrate why the rule (4) cannot be applied to uniqueness constraints: Suppose, there are two extensions project group 1 and project group 2. On each of these extensions there is a uniqueness constraint defined. The employees which are involved in the project group 1 are unambiguously identified by their social security number whereas the employees of project group 2 are distinguished using their names. How can these original integrity constraints be handled after unifying the corresponding class extensions? Obviously, we cannot assume that one of the original constraints holds for the whole united extension. On the other hand, these constraints cannot be skipped because of the requirement that the federated schema has to be complete in order to maintain the semantics of the original schemata to be integrated.

This problem can be solved by using *discriminants* [GCS95,AQFG95]. Discriminants can be used to maintain the relationship of the objects of the united extension with the original class extensions they stem from. For the example described above we can define a discriminant attribute group which can be used to specify integrity constraints on the united extension. For instance, the following integrity constraints can be derived in our example: All employee objects whose discriminant has the value 'project group 1' must have different social security numbers. Analogously, all employee objects whose discriminant has the value 'project group 2' must have different names.

4.2 Referential Integrity Constraints

Here, we restrict the discussion of the applicability of the rules for referential integrity constraints to constraints which refer to object references within a single

extension. In our example, there may be a referential integrity constraint on the class extension employees which defines that the attribute supervisor must refer to an existing employee object.

We state that the rule (1) does not hold for such kinds of constraints. This is due to the fact that the result of a selection operation is a subset of the original extension, i.e., some of the objects of the original extension (which do not fulfill the selection condition) are not in the resulting extension. Thus, originally valid references can now be undefined. In other words, we cannot assume that a referential integrity constraint holds for the resulting extension if potentially referenced objects are not in the resulting extension. The rules (2) and (3) do not hold for the same reason.

In contrast, the union rule (4) holds for referential integrity constraints. The reason is that all references of the original extension stay valid because all objects of this extension are in the united extension. Suppose, we unite the extensions project group 1 and project group 2 to the extension employee. Here, we assume that all references are valid because all referenced employees are in the united extension. However, we have to notice that the logical disjunction of the two original integrity constraints leads to a weakening of the constraints. As discussed before, a discriminant approach can be used to solve this problem.

4.3 Aggregate Integrity Constraints

Finally, we discuss the applicability of the rules to aggregate constraints defined by using the aggregate functions max and min. Here, we consider constraints of the form 'max(x) θ z' and 'min(x) θ z', respectively, whereby x is an attribute, z a constant value and θ is a comparison operator.

In general, the applicability of the rules to such kinds of constraints[2] depends on the comparison operator θ used for the formulation of such kinds of constraints.

For max-constraints it can be easily shown that the rules (1), (2) and (3) can only be applied if the following holds for the comparison operator: $\theta \in \{\leq, <\}$. If the maximum of an attribute x of an extension E is less (or less equal) than a certain value z, then the maximum of the attribute x of each subset of the extension E is also less (or less equal) than the value z. However, in case a max-constraint is defined with one of the following comparison operators $\{\geq, >, =, \neq\}$, then we cannot guarantee that the maximum of the resulting extension fulfills this constraint. Suppose, there is an extension employees on which the max-constraint 'max(salary) > 4000' is defined. Here, it can happen that the object which make the condition valid in the original extension is not available in the resulting extension. Obviously, this is the case when, for instance, the selection condition is 'max(salary) $\leq$ 4000'.

Analogously, the rules (1), (2) and (3) can only be applied to min-constraints if the comparison operator θ is in $\{\geq, >\}$.

The rule (4) is obviously applicable to max-constraints if the comparison operator θ is in $\{\geq, >\}$ and for min-constraints if the comparison operator θ is in $\{\leq, <\}$.

[2] In the following, aggregate constraints based on the function max are called max-constraints. Analogously, aggregate constraints based on the function min are called min-constraints.

Assume, there is a constraint 'max(salary) $\geq$ 4000' defined on a class extension. Then, we can assume that this constraint also holds for union with another extension because this constraint is already fulfilled by the original extension. In this case, union means that additional objects (of a second extension) are put in this extension. Therefore, the maximum salary of the united extension can only be greater or equal than the maximum salary of the original extension.

Analogously, assume that there is a constraint 'min(salary) $\leq$ 2000' defined on a class extension. This constraint also holds for union because the minimum salary of the united extension can only be lower or equal than the minimum salary of the original extension.

However, there are two other cases where rule (4) is applicable, too. Firstly, it holds for max-constraints if the comparison operator θ is in $\{\leq, <\}$. Secondly, this rule can be applied to min-constraints if the comparison operator θ is in $\{\geq, >\}$. The reason for the applicability in both cases is that these kinds of constraints can be transformed into object constraints. The class constraint 'max(x) θ_1 z' with $\theta_1 \in \{\leq, <\}$ is equivalent to the object constraint 'x θ_1 z' and the class constraint 'min(x) θ_2 z' with $\theta_2 \in \{\geq, >\}$ corresponds to the object constraint 'x θ_2 z'. Due to the fact that the rule (4) is applicable to any object constraint we can conclude that this rule can also be applied to these special kinds of max- and min-constraints.

As discussed before, a discriminant is required in order to avoid the weakening of the constraints by applying rule (4). However, the results of the applicability of the rules to class constraints are summarized in the following table:

Rule	unique	ref	$\max_\theta$	$\min_\theta$
(1) *selection*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(2) *difference*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(3) *intersection*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(4) *union*	—	$\surd$	$\theta \in \{\geq, >, <, \leq\}$	$\theta \in \{\leq, <, \geq, >\}$

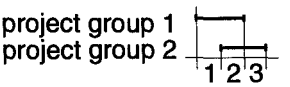
5 Example for Rule Application

In this section we demonstrate and discuss the application of our proposed rules by means of a small example. For the presentation, we distinguish two phases:

1. decomposing the original extensions into base extensions, and
2. composing these base extensions by uniting them to derived extensions.

5.1 Decomposing Extensions

The decomposition of the original class extensions is based on an extensional analysis. Such an analysis specifies in which way we have to construct the base extensions from the original extensions by applying set operations. In our example, we start with two class extensions E_1 and E_2 which contain employees working in two different projects groups. We assume that the result of the extensional analysis is as it is depicted in the following figure:



There may be employees working only for one of the two projects, and there are employees working for both projects at the same time. In consequence, the class extensions overlap.
Now, we decompose the original class extensions into three base extensions. The decomposition can be described using the set operations intersection and difference between E_1 and E_2:

$$E_1' = E_1 - E_2, \quad E_2' = E_1 \cap E_2, \quad E_3' = E_2 - E_1$$

In Figure 2 the resulting three base extensions are placed on the second level above the two original class extensions.

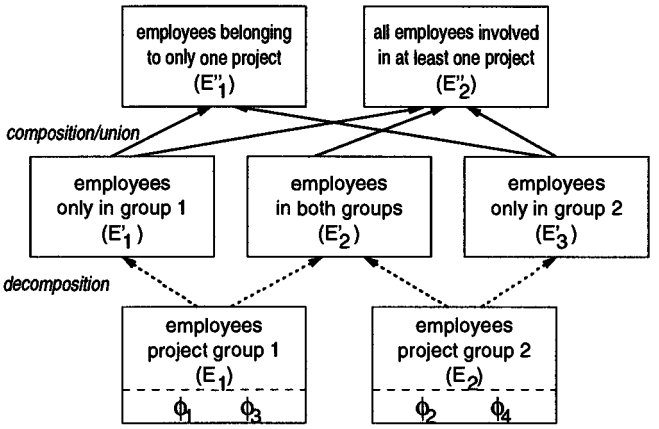


Fig. 2. Decomposing and uniting extensions of the example

Now, let us assume that we have the following integrity constraints imposed on the two original class extensions:

$$\Theta_{\phi_1}(E_1) \; with \; \phi_1 \equiv \forall e : e.\text{qualification} > 4$$
$$\Theta_{\phi_3}(E_1) \; with \; \phi_3 \equiv \forall e_1, e_2 : e_1.\text{pid} = e_2.\text{pid} \Longrightarrow e_1 = e_2$$
$$\Theta_{\phi_2}(E_2) \; with \; \phi_2 \equiv \forall e : e.\text{qualification} > 3$$
$$\Theta_{\phi_4}(E_2) \; with \; \phi_4 \equiv \exists e_1 : \forall e_2 : e_1.\text{salary\_level} \geq e_2.\text{salary\_level} \wedge e_1.\text{salary\_level} > 3$$

ϕ_1 and ϕ_2 are object constraints whereas ϕ_3 and ϕ_4 are class constraints. ϕ_3 requires the uniqueness of values for the attribute pid (i.e., pid can be used as a key). ϕ_4 is an aggregate constraint concerning the maximum value for the attribute salary_level (we may use the short term max(salary_level)>3). In other words there must be at least one employee of extension E_2 having a higher salary level than 3.

Using the relationships between base extensions and original class extensions we can derive integrity constraints for the base extensions. For that we have to

consider each constraint and to apply our rules. The object constraints ϕ_1 and ϕ_2 do not cause any problem. By applying rule (2) and (9) we derive:

$$\Theta_{\phi_1}(E_1) \xrightarrow{\text{rule 9}} \Theta_{\phi_1}(E_1' \cup E_2') \xrightarrow{\text{rule 2}} \Theta_{\phi_1}(E_1')$$
$$\Theta_{\phi_1}(E_1) \xrightarrow{\text{rule 9}} \Theta_{\phi_1}(E_2' \cup E_1') \xrightarrow{\text{rule 2}} \Theta_{\phi_1}(E_2')$$
$$\Theta_{\phi_2}(E_2) \xrightarrow{\text{rule 9}} \Theta_{\phi_2}(E_2' \cup E_3') \xrightarrow{\text{rule 2}} \Theta_{\phi_2}(E_2')$$
$$\Theta_{\phi_2}(E_2) \xrightarrow{\text{rule 9}} \Theta_{\phi_2}(E_3' \cup E_2') \xrightarrow{\text{rule 2}} \Theta_{\phi_2}(E_3')$$

Transferring the class constraints from the original extensions to the base extensions is more complicated. For instance, the application of rule (2) to ϕ_3 leads to a weakening of the constraint. As long as we do not forget the original constraint, this is not a real problem:

$$\Theta_{\phi_3}(E_1) \xrightarrow{\text{rule 9}} \Theta_{\phi_3}(E_1' \cup E_2') \xrightarrow{\text{rule 2}} \Theta_{\phi_3}(E_1')$$
$$\Theta_{\phi_3}(E_1) \xrightarrow{\text{rule 9}} \Theta_{\phi_3}(E_2' \cup E_1') \xrightarrow{\text{rule 2}} \Theta_{\phi_3}(E_2')$$
$$\Theta_{\phi_3}(E_1) \xrightarrow{\text{rule 9}} \Theta_{\phi_3}(E_1' \cup E_2')$$

In contrast to the uniqueness constraint ϕ_3, the max-constraint ϕ_4 cannot be transferred to any base extension. Hence, only rule (9) can be applied:

$$\Theta_{\phi_4}(E_2) \xrightarrow{\text{rule 9}} \Theta_{\phi_4}(E_2' \cup E_3')$$

5.2 Uniting Extensions

For the decomposition of extensions the respective rules are applied to each *single* constraint. However, for the composition the application of rule (4) to each simple constraint is not useful. In order to avoid a constraint weakening, rule (4) has to be applied to *all* possible combinations of constraints and afterwards the results have to be conjunctively combined by rule (7). A simpler way to deal with constraints is to conjunctively combine the constraints of each base extension before applying other rules.

In our integration example the federated schema consists of the class extensions E_1'' and E_2''. Extension E_1'' results from uniting base extensions E_1' and E_3' and contains all employees working in only one project group whereas extension E_2'' contains all employees. The class E_1'' is a subclass of class E_2''. This federated schema is only one possible federated schema. However, it demonstrates the application of our proposed rules.

The constraints of base extensions E_1' and E_2' have to be combined by applying rule (7). On base extension E_3' only one constraint holds. Therefore, the constraints of the base extensions E_1' and E_2' only have to be combined:

$$\Theta_{\phi_1}(E_1'), \Theta_{\phi_3}(E_1') \xrightarrow{\text{rule 7}} \Theta_{\phi_1 \wedge \phi_3}(E_1')$$
$$\Theta_{\phi_1}(E_2'), \Theta_{\phi_2}(E_2'), \Theta_{\phi_3}(E_2') \xrightarrow{\text{rule 7}} \Theta_{\phi_1 \wedge \phi_2 \wedge \phi_3}(E_2') \leftrightarrow \Theta_{\phi_1 \wedge \phi_3}(E_2')$$

Here, constraint ϕ_2 can be omitted since it is weaker than constraint ϕ_1.

$\mathbf{E_1''}$: The rule (4) disjunctively brings together the combined constraints of the base extensions E_1' and E_3':

$$\Theta_{\phi_1 \wedge \phi_3}(E_1'), \Theta_{\phi_2}(E_3') \xrightarrow{\text{rule } 4} \Theta_{(\phi_1 \wedge \phi_3) \vee \phi_2}(E_1' \cup E_3') \leftrightarrow \Theta_{\phi_2}(E_1' \cup E_3') \xrightarrow{\text{rule } 9} \Theta_{\phi_2}(E_1'')$$

Due to the fact that $\phi_1 \Rightarrow \phi_2$ holds, the formula $(\phi_1 \wedge \phi_3) \vee \phi_2$ can be simplified to ϕ_2. Neither constraint ϕ_3 nor ϕ_4 holds on E_1''. Objects can be inserted into extension E_1'' which cannot be propagated to E_1 nor E_3 because both class constraints are violated. It is impossible to test these class constraints for extension E_1'' due to the missing base extension E_2' on the composition level. Errors from violating update operations cannot be interpreted on this level. This problem is similar to the well-known *view-update*-problem of selective views.

$\mathbf{E_2''}$: The extension E_2'' can be formed by:

$$E_2'' = E_1' \cup E_2' \cup E_3' = (E_1' \cup E_2') \cup (E_2' \cup E_3')$$

The integrity constraints of class extension $E_1' \cup E_2'$ cannot be derived directly by applying rule (4) due to the class constraint ϕ_3. This constraint has to be weakened before:

$$\Theta_{\phi_1 \wedge \phi_3}(E_1') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_1') \qquad \Theta_{\phi_1 \wedge \phi_3}(E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_2')$$

The application of rule (4) produces:

$$\Theta_{\phi_1}(E_1'), \Theta_{\phi_1}(E_2') \xrightarrow{\text{rule } 4} \Theta_{\phi_1 \vee \phi_1}(E_1' \cup E_2') \leftrightarrow \Theta_{\phi_1}(E_1' \cup E_2')$$

By applying rule (7) we obtain:

$$\Theta_{\phi_1}(E_1' \cup E_2'), \Theta_{\phi_3}(E_1' \cup E_2') \xrightarrow{\text{rule } 7} \Theta_{\phi_1 \wedge \phi_3}(E_1' \cup E_2')$$

Analogously to $E_1' \cup E_2'$, we have to weaken the constraint before applying rule (4):

$$\Theta_{\phi_1 \wedge \phi_3}(E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_2')$$
$$\Theta_{\phi_1}(E_2'), \Theta_{\phi_2}(E_3') \xrightarrow{\text{rule } 4} \Theta_{\phi_1 \vee \phi_2}(E_2' \cup E_3') \leftrightarrow \Theta_{\phi_2}(E_2' \cup E_3')$$

Due to the fact that $\phi_1 \Rightarrow \phi_2$ holds, the formula $\phi_1 \vee \phi_2$ can be simplified to ϕ_2. By applying rule (7) we obtain:

$$\Theta_{\phi_2}(E_2' \cup E_3'), \Theta_{\phi_4}(E_2' \cup E_3') \xrightarrow{\text{rule } 7} \Theta_{\phi_2 \wedge \phi_4}(E_2' \cup E_3')$$

Before the extension $E_1' \cup E_2'$ and $E_2' \cup E_3'$ can be united by applying rule (4) the constraint ϕ_3 has to be removed:

$$\Theta_{\phi_1 \wedge \phi_3}(E_1' \cup E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_1' \cup E_2')$$

By applying rule (4) we obtain:

$$\Theta_{\phi_1}(E_1' \cup E_2'), \Theta_{\phi_2 \wedge \phi_4}(E_2' \cup E_3') \xrightarrow{\text{rule } 7} \Theta_{\phi_1 \vee (\phi_2 \wedge \phi_4)}(E_1' \cup E_2') \xrightarrow{\text{rule } 9} \Theta_{\phi_1 \vee (\phi_2 \wedge \phi_4)}(E_2'')$$

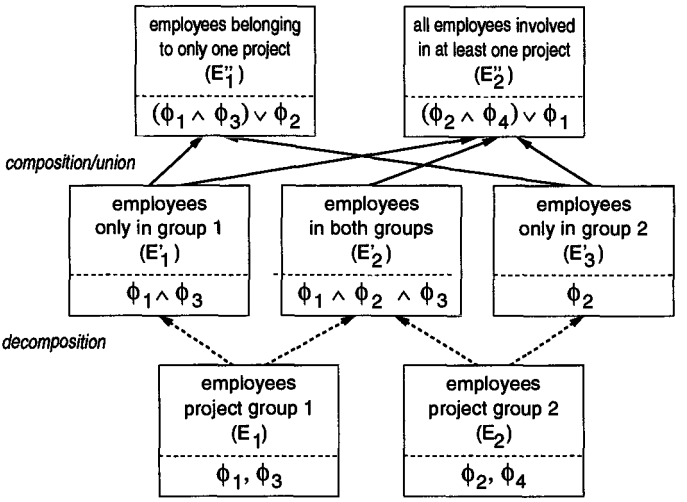


Fig. 3. Extensions with derived integrity constraints

Figure 3 shows the extensions with their integrity constraints in the unreduced form.

An interesting point is the fact that the uniqueness constraint ϕ_3 disappeared from extension E_1'' due to the disjunctive combination expressed in rule (4). The uniqueness constraint does not hold for the extension E_2'', too. The introduction of an artificial discriminant attribute avoids such an disappearance. The discriminant attribute (e.g. group) in our example expresses the information whether an employee works for project group 1 (value 1) or project group 2 (value 2) or for both of them (value 3). This information can be used within the constraints in form of an implication. The left hand side of the implication tests the discriminant attribute on a specific value and the right hand side contains the original constraint. We transform the constraints ϕ_1, ϕ_2, ϕ_3, and ϕ_4 to $\phi_1', \phi_2', \phi_3'$, and ϕ_4', respectively, and assign them to the extensions E_1', E_2', and E_3'.

$$\phi_1' \equiv \forall e : (e.\text{group} = 1 \vee e.\text{group} = 3) \Longrightarrow e.\text{qualification} > 4$$

$$\phi_2' \equiv \forall e : (e.\text{group} = 2 \vee e.\text{group} = 3) \Longrightarrow e.\text{qualification} > 3$$

$$\phi_3' \equiv \forall e_1, e_2 : ((e_1.\text{group} = 1 \vee e_1.\text{group} = 3) \wedge (e_2.\text{group} = 1 \vee e_2.\text{group} = 3)) \Longrightarrow (e_1.\text{pid} = e_2.\text{pid} \Longrightarrow e_1 = e_2)$$

$$\phi_4' \equiv \exists e_1 : \forall e_2 : ((e_1.\text{group} = 2 \vee e_1.\text{group} = 3) \wedge (e_2.\text{group} = 2 \vee e_2.\text{group} = 3)) \Longrightarrow e_1.\text{salary\_level} \geq e_2.\text{salary\_level} \wedge e_1.\text{salary\_level} > 3$$

$$\Theta_{\phi_1' \wedge \phi_3'}(E_1'), \qquad \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_2'), \qquad \Theta_{\phi_2'}(E_3')$$

These transformed constraints can now be conjunctively combined. The conjunctive combination stands in contrast to rule (4).

$$\Theta_{\phi_1' \wedge \phi_3'}(E_1'), \Theta_{\phi_2'}(E_3') \longrightarrow \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_1' \cup E_3') \xrightarrow{\text{rule 9}} \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_1'')$$

$$\Theta_{\phi'_1 \wedge \phi'_3}(E'_1), \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3}(E'_2), \Theta_{\phi'_2}(E'_3), \Theta_{\phi'_4}(E'_2 \cup E'_3)$$
$$\longrightarrow \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4}(E'_1 \cup E'_2 \cup E'_3) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4}(E''_2)$$

Introducing discriminant attributes and respective transformation of constraints reduce weakening of constraints during schema integration (or more general during schema restructuring).

6 Conclusions

The complete treatment of integrity constraints during the schema integration is a fundamental requirement, e.g., for global integrity enforcement in federated database systems. Moreover, the extensive consideration of integrity constraints plays an important role in application scenarios such as view materialization [SJ96] or mobile databases [WSD+95].

In this paper, we presented an approach to handle integrity constraints during the schema integration. We found a small set of general rules which specify how to deal with integrity constraints during the integration process. We showed that these rules can be used for object constraints without any restrictions. For several classes of often occurring class constraints we presented the applicability of the rules. Of course, there are some logical restrictions for which specialized solutions are needed. Beside the results presented here, we have already investigated further classes of constraints, e.g. constraints with other aggregate functions.

Furthermore, we presented a systematic application of the rules. By means of a running example we sketched some critical problems such the weakening of constraints and demonstrated how to handle these kinds of problems (e.g. by using discriminants).

Our approach is not limited to schema integration purposes only, it can also be used for view derivation and arbitrary schema modification operations such as schema transformation or schema restructuring. We are currently working on the completion of the classification of integrity constraints and the applicability of our rules to further classes of constraints. Using a classification which is based on the logical structure of constraints (assuming an adequate normal form for the logical representation of constraints) seems to be a very promising approach for completing the formal part of our work.

Finally, we can state that beside our schema integration strategies presented in [SS96a,SS96b,SC97], the consideration of integrity constraints is another step towards a unified integration methodology.

References

[AQFG95] R. M. Alzahrani, M. A. Qutaishat, N. J. Fiddian, and W. A. Gray. Integrity Merging in an Object-Oriented Federated Database Environment. In C. Goble and J. Keane, eds., *Advances in Databases, Proc. BNCOD-13*, LNCS 940, pages 226–248. Springer, 1995.

[BC86] J. Biskup and B. Convent. A Formal View Integration Method. In C. Zaniolo, ed., *Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, 15(2):398–407, 1986.

[BCW91] M. Bouzeghoub and I. Comyn-Wattiau. View Integration by Semantic Unification and Transformation of Data Structures. In H. Kangassalo, ed., *Entity-Relationship Approach: The Core of Conceptual Modelling, Proc. ER'90,* pages 381–398. North-Holland, 1991.

[CHS+97] S. Conrad, M. Höding, G. Saake, I. Schmitt, and C. Türker. Schema Integration with Integrity Constraints. In C. Small, P. Douglas, R. Johnson, P. King, and N. Martin, eds., *Advances in Databases, Proc. BNCOD-15*, LNCS 1271, pages 200–214. Springer, 1997.

[Con86] B. Convent. Unsolvable Problems Related to the View Integration Approach. In G. Ausiello and P. Atzeni, eds., *Proc. 1st Int. Conf. Database Theory (ICDT'86)*, LNCS 243, pages 141–156. Springer, 1986.

[EJ95] L. Ekenberg and P. Johannesson. Conflictfreeness as a Basis for Schema Integration. In S. Bhalla, ed., *Information Systems and Data Management, Proc. CISMOD'95*, LNCS 1006, pages 1–13. Springer, 1995.

[GCS95] M. Garcia-Solaco, M. Castellanos, and F. Saltor. A Semantic-Discriminated Approach to Integration in Federated Databases. In S. Laufmann, S. Spaccapietra, and T. Yokoi, eds., *Proc. 3rd Int. Conf. on Cooperative Information Systems (CoopIS'95)*, pages 19–31, 1995.

[LNE89] J. A. Larson, S. B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989.

[RPG95] M. P. Reddy, B. E. Prasad, and A. Gupta. Formulating Global Integrity Constraints during Derivation of Global Schema. *Data & Knowledge Engineering*, 16(3):241–268, 1995.

[SC97] I. Schmitt and S. Conrad. Restructuring Class Hierarchies for Schema Integration. In R. Topor and K. Tanaka, eds., *Database Systems for Advanced Applications '97, Proc. DASFAA'97*, pages 411–420, World Scientific, 1997.

[SJ96] M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 75–86. Morgan Kaufmann, 1996.

[SL90] A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990.

[SPD92] S. Spaccapietra, C. Parent, and Y. Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal*, 1(1):81–126, 1992.

[SS96a] I. Schmitt and G. Saake. Integration of Inheritance Trees as Part of View Generation for Database Federations. In B. Thalheim, ed., *Conceptual Modelling — ER'96*, LNCS 1157, pages 195–210. Springer, 1996.

[SS96b] I. Schmitt and G. Saake. Schema Integration and View Generation by Resolving Intensional and Extensional Overlappings. In K. Yetongnon and S. Hariri, eds., *Proc. 9th ISCA Int. Conf. on Parallel and Distributed Computing Systems (PDCS'96)*, pages 751–758. International Society for Computers and Their Application, Releigh, NC, 1996.

[VA96] M. W. W. Vermeer and P. M. G. Apers. The Role of Integrity Constraints in Database Interoperation. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 425–435. Morgan Kaufmann, 1996.

[WSD+95] O. Wolfson, P. Sistla, S. Dao, K. Narayanan, and R. Raj. View Maintenance in Mobile Computing. *ACM SIGMOD Record*, 24(4):22–27, 1995.

Scoped Referential Transparency in a Functional Database Language with Updates

P.F.Meredith and P.J.H King

Dept. of Computer Science, Birkbeck College, Malet St, London WC1E 7HX
{P.F.Meredith@btinternet.com, pjhk@dcs.bbk.ac.uk}

Abstract. We describe in brief a lazy functional database language *Relief*, which supports an entity-function model and provides for update and the input of data by means of functions with side-effects. An *eager let* construct is used within the lazy graph reduction mechanism to sequence the effects. To redress the loss of referential transparency we have implemented an effects checker which can identify referentially transparent regions or scopes within *Relief*.

1 Introduction

An inherent conflict arises with functional languages in the database context between change of state on update and the concept of referential transparency. Passing the database itself as a parameter resolves the issue conceptually but is not a practical approach. Update by side-effect is efficient but necessarily implies loss of referential transparency. Additionally side-effects are often incompatible with lazy evaluation since the occurrence and sequence of updates can become non-intuitive. Our approach is to accept that a change of database state will result in loss of global referential transparency but that referential transparency can be scoped so that areas of a program can be identified which retain the property. Equational reasoning and consequent optimisation can then be applied within these scopes.

We describe a variant of the functional database language FDL [Pou88, PK90], called *Relief*, which is lazy and which has been extended with functions to facilitate updates and reading of files. These functions work by means of side-effects which are explicitly sequenced with an eager let expression and a derivative sequencing construct. There is no change in the overall laziness of the graph reduction which we illustrate with an example update program which employs the lazy stream model. To redress the loss of global referential transparency an effects checker identifies referentially transparent regions or scopes.

2 The Entity-Function Model

The development of *Relief* is a contribution to the work of the Tristarp Group, initiated in 1984 to investigate the feasibility of a database management system which

supports the binary relational model both at the storage and conceptual levels. A summary of the early work is given by King et al. [KDPS90]. More recently research has focused on supporting the entity-function model, a hybrid of the binary relational and functional data models [Shi81, AK94]. The relationships between entities are binary and are represented by (possibly) multivalued single argument functions. Entity types are classified into lexical and non-lexical as with the analysis methodology NIAM [VV82]. Members of lexical types have direct representation (for example the types string and integer) whereas those of non-lexical types are real or abstract objects without direct representation eg. persons, invoices, courses etc.

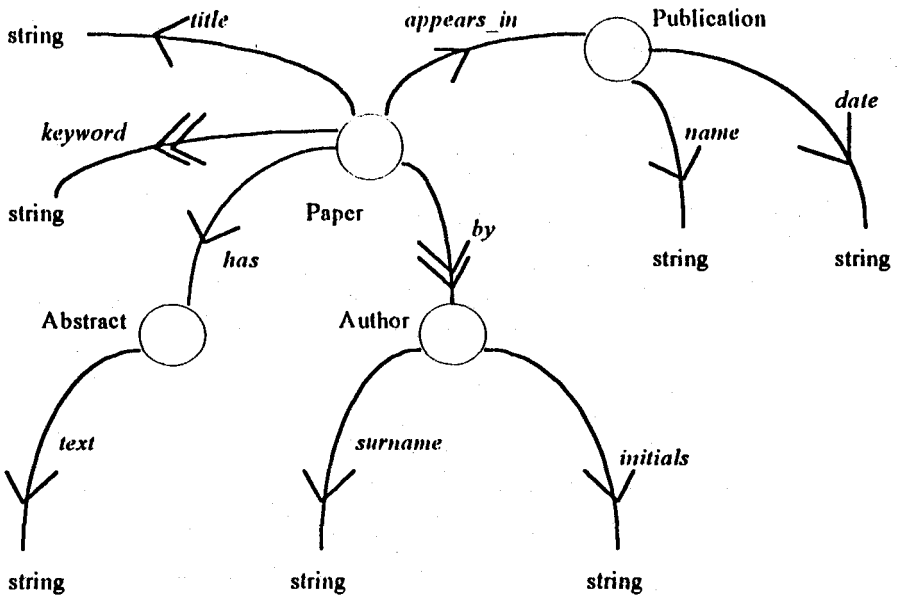


Fig. 1. The Computer Literature Database (CLDB)

Figure 1 is a subschema of the Computer Literature Database (CLDB), a body of data used experimentally by the Tristarp Group, and illustrates the entity-function model. Non-lexical types are shown as circles with functions represented by arcs. A single-headed arrow represents a single-valued function and a double-headed one denotes a multivalued function. Thus a Paper has a single Abstract but is *by* (possibly) several Authors. In this example, the lexical type "string" is used extensively in the ranges of the functions.

3 Defining a Database in *Relief*

Relief is an experimental variant of FDL [Pou88, PK90], the first language to integrate the entity-function model with the functional computation model over a

binary relational storage structure. FDL has the usual features of a modern functional programming language such as polymorphic types, lazy evaluation and pattern-matching and we refer the reader to Field and Harrison [FH88] for an explanation of these. We describe here only the features of *Relief* which illustrate the scoping of referential transparency. A full syntax and description of the language and its implementation is given by Meredith [Mer98]. The persistent functions and entities which constitute a database in *Relief* are held in a triple store, a development of the grid file [NHS84, Der89] which has recently been adapted to hold long strings and to support interval queries.

CLDB contains the non-lexical types *paper* and *author* amongst others. This part of the schema is implemented in *Relief* with the following type declarations:

```
paper :: nonlex;
author :: nonlex;
by : paper -> [author];
title : paper -> string;
surname : author -> string:
initials : author -> string;
```

In order to facilitate easy navigation of the schema *Relief* automatically maintains a converse function for all functions whose domain is a non-lexical type, referred to by the prefix *rev*. For example the converses of the functions *surname* and *by* are:

```
rev_surname : string -> [author];
rev_by : author -> [ paper ];
```

As an example of their use consider the following list comprehension query which retrieves the papers by authors with the surname "Meredith".

```
[ title p | for a in rev_surname "Meredith";
  for p in rev_by a] ;
```

It is interesting to note here the syntactic similarity between *Relief* and the "FOR EACH" statement in DAPLEX [Shi81].

4 The Four Update Functions of *Relief*

The extent of a database is defined and can be redefined by means of the functions *create*, *destroy*, *define* and *delete*. The function *create* takes a non-lexical type as an argument and returns a new entity of the specified type. For instance:

```
create author;
```

returns a new member of the type *author*. An attempt to print a non-lexical entity yields its type in angled brackets, thus: <*author*>. The extent of a non-lexical type

can be queried with the *All* function which returns the members of the specified type. For example, `All author ;`

The function *destroy* destroys an entity of the specified non-lexical type. Thus to destroy the first element in the list of all authors:

```
destroy author (head (All author));
```

In the functional model it is necessary for every expression to have a value and thus destroy returns the value *Done* which is the only value in the type *update*. Functions are incrementally defined using the function *def* which adds an equation to the database and returns the value *Done*. To illustrate this, we give an expression which creates an author entity, binds it to a scoped static variable "a" and then defines an equation for their initials. The binding is done with an eager form of the let expression which is further explained in the next section. Thus:

```
e_let a = create author in (def initials a => "PF");
```

The left hand side of a function equation consists of the name of the function followed by one or more arguments, and the right hand side is an expression which is stored unevaluated. The arguments at definition time can either be patterns or a bound variable such as "a", in which case the value of the variable is used in the definition of the equation. The value of a function's application defaults to the constant "@" (denoting "undefined") if there is no equation which matches the function's arguments. The pattern-matching in *Relief* uses the left-to-right best-fit algorithm developed by Poulovasillis [Pou92].

The delete function *del* deletes an equation from the database and returns the value *Done*. The arguments of *del* identify the left hand side of the function equation for deletion, for example:

```
e_let a = head(rev_surname "King") in (del initials a);
```

deletes the equation defining the initials of an author with the surname "King".

All four update functions are statically type checked, see Meredith [Mer98] for details, and work at the object level. Thus their invocation needs to be sequenced and we discuss this next.

5 Sequencing the Updates

A new construct, the *eager let* (e_let), is introduced to sequence updates by employing call-by-value parameter substitution but within a general scheme of lazy evaluation. We illustrate the eager let expression with the following example which evaluates an expression before using it as the value of the right hand side in an equation definition. The example assumes that people have a unique name and the types of the functions *name* and *salary* are:

```
name : person -> string;
salary : person -> integer;
```

The expression below defines the salary of Bill to be 5000 more than Fred's current salary.

```
e_let bill = head(rev_name "Bill") in
    e_let fred = head(rev_name "Fred") in
          e_let bills_salary = salary fred + 5000 in
                (def salary bill => bills_salary );
```

Thus the value of Bill's salary is stored as an integer and not a formula and so a subsequent change to Fred's salary will not result in a change to Bill's salary. The next example produces a sequence of updates to create an author entity and define *surname* and *initials* before returning the new entity. Two dummy variables, *dum1* and *dum2*, are used to bind with the values returned by calls to the define function.

```
e_let a = create author in
    e_let dum1 = (def surname a => "Meredith") in
          e_let dum2 = (def initials a => "PF")in a;
```

The overall evaluation scheme is still normal order reduction, the outermost let being reduced before the innermost let and the body of each eager let being evaluated lazily. The syntax of the eager let is not ideal because of the need to specify binding patterns which are of no further interest. Hence we have provided an explicit sequencing construct in *Relief* which, although formally only syntactic sugar for the eager let, is much more succinct and convenient for programming. This construct is written as a list of expressions separated by semicolons, enclosed in curly brackets, and is optionally terminated by the keyword *puis* and a following expression which gives a functional value for the entire sequence. The expressions between the brackets are evaluated in their order, from left to right. For example, the expression above can be written as the sequence:

```
 { a = create author; (def surname a=>"Meredith");
(def initials a => "PF") } puis a;
```

We refer to the keyword puis and its following expression as the *continuation*. If it is omitted then a default value of *Done* is inferred for the sequence.

6 An Example - Updating the Computer Literature Database

We describe here a *Relief* program which processes a list of data constructors representing the records in a file "Tristarp_papers" and updates that part of the CLDB concerning authors and papers. *Relief* contains a *read_file* function which can lazily convert a file of records into such a list but space precludes a full description here.

The elements of the list are members of the *paper_record* sum type which is defined by the two constructors TITLE_REC and AUTHOR_REC. TITLE_REC has a single string component, whereas AUTHOR_REC has two components: one for the surname of the author and one for the initials:

```
paper_record :: sum;
TITLE_REC : string -> paper_record;
AUTHOR_REC : string string -> paper_record;
```

Our example will assume the following list of records as input:
[(TITLE_REC "Scoped Referential Transparency in a Functional Database Language with Updates"), (AUTHOR_REC "Meredith" "PF"), (AUTHOR_REC "King" "PJH"), (TITLE_REC "TriStarp - An Investigation into the Implementation and Exploitation of Binary Relational Storage Structures"), (AUTHOR_REC "King" "PJH"), (AUTHOR_REC "Derakhshan" "M"), (AUTHOR_REC "Poulovassilis" "A"), (AUTHOR_REC "Small" "C")]

The program to process the list consists of two functions: *cldbup* which creates the paper entities, and *add_authors*, which creates the author entities and links them to the papers. Both functions operate recursively and process records at the head of an input list before returning the remainder of the list. For a list with a title record at its head, cldbup creates a paper, defines its title and calls add_authors as specified by the first equation below. If the head of the list is not a title record then the list is returned unaltered as given by the second equation. This is an appropriate action when an unexpected or unrecognised record is encountered. The third equation specifies that cldbup returns the empty list when given the empty list. Thus cldbup will return the empty list if all of the records in the file can be processed, otherwise it returns some remainder of the input list.

```
cldbup : [paper_record] -> [paper_record];
def cldbup ((TITLE_REC t) : xs) => {
    p = create paper;
    (def title p => t)
    } puis cldbup (add_authors p xs);
def cldbup (h : t) => (h : t);
def cldbup [ ] => [ ];
```

The function *add_authors* recursively processes all the author records at the front of the list before returning the remainder of the list. The logic for processing a single author record is:

- construct a list of authors with the specified surname and initials
- if the list is empty create a new author, define their surname and initials, and bind them to the pattern "a" else bind the pattern to the head of the list of authors
- if the list of authors for the specified paper is currently undefined, then define a list with the author "a" as its single member. Otherwise add the author "a" to the existing list.

```
add_authors : paper [paper_record] -> [paper_record];
```

```
def add_authors p ((AUTHOR_REC s i) : xs) => {
    auth_list = [ a1 | for a1 in rev_surname s;
                                        initials a1 == i ];
    a = if auth_list == [ ] then
          {a2 = create author;(def surname a2 => s);
                        (def initials a2 => i) } puis a2
        else head auth_list fi;
    e_let authors = by p in
          if authors == @ then (def by p => [a])
          else e_let new_authors = append authors [a]
                in (def by p => new_authors)
          fi
    } puis add_authors p xs;
def add_authors p (h : t) => (h : t);
def add_authors p [ ] =>[ ];
```

The file of papers is then processed lazily by evaluating the following expression:

```
cldbup (read_file paper_record "Tristarp_papers");
```

7 Scoping Referential Transparency

A defining property of languages which are purely functional is said to be *referential transparency* [Hud89]. It is this property which allows functional languages to be declarative rather than imperative in nature. Put simply, referential transparency means that every instance of a particular variable has the same value within its scope of definition. This is the interpretation of variables in algebra and logic which allows the use of equational reasoning.

Although *Relief* is not a pure functional language it is possible to identify regions of referential transparency where the techniques used in functional programming, and by functional compilers, can still be applied. This follows from the fact that the concept of referential transparency is scoped by definition. In a purely functional language the scope is the environment or script in which an expression is evaluated. Referential transparency in *Relief* is scoped by the update operations which destroy reference. These regions of referential transparency can be statically identified by means of an *effects checker* which classifies the effect of an expression. Furthermore an effects checker is of use in concurrent systems since it identifies potential interference between expressions.

7.1 Gifford and Lucassen's Effects Checker

Effects checking was first described by Gifford and Lucassen [GL86] in connection with their research into the integration of imperative and functional languages. Gifford and Lucassen identify the following three kinds of effects of expressions:

- A – allocation and initialisation of mutable storage, for example the declaration of a variable or collection.
- R – reading a value from mutable storage.
- W – writing to mutable storage, that is assignment.

There are eight possible combinations of these three effects which form a lattice under the subset relation. These eight different combinations can be grouped into effect classes depending on the language requirement. Gifford and Lucassen were interested in identifying those parts of a program which could be memoised and were amenable to concurrent evaluation. Hence they produced an effect checking system which classified expressions into one of the following four effect classes:

{ W } { W,R } { W,A} { W,R,A }	- PROCEDURE class
{ R } { R, A }	- OBSERVER class
{ A }	- FUNCTION class
{ }	- PURE class

There is a total ordering of these effect classes in the sense that they form a hierarchy of effect properties. The effect properties of the four classes can be summarised as:

- PROCEDURE – expressions of this sublanguage can define variables, read mutable storage and write mutable values.
- OBSERVER – expressions of this sublanguage can define variables and read variables. They cannot perform assignments and cannot make use of expressions with the effect class PROCEDURE.
- FUNCTION – expressions of this sublanguage can define variables but they cannot read variables or change their value. FUNCTION expressions cannot make use of PROCEDURE or OBSERVER expressions.
- PURE – a pure function has no effects and cannot be affected by the evaluation of other expressions. It can neither define variables, nor read, nor write them. A PURE expression cannot make use of PROCEDURE, OBSERVER or FUNCTION expressions.

The only expressions which can interfere with concurrency are those in the effect class PROCEDURE and these can only interfere with other expressions in the PROCEDURE and OBSERVER effect classes. Thus the ability to identify expressions of these classes would be important in order to maximise concurrency in a multi-user system. In contrast PURE, FUNCTION and OBSERVER expressions do not interfere with each other and can be run concurrently. PURE expressions can be memoised and OBSERVER expressions can be memoised between updates. We now look at how effects checking is applied in *Relief*.

7.2 The Effects Checker in *Relief*

We begin by classifying the effect properties of constructs in *Relief*. *Relief* has no updateable variables in the usual sense but it does have extensible and updateable

types, namely user-defined functions and non-lexical entities. Thus the Gifford and Lucassen concept of the allocation of mutable storage is equivalent to the declaration of such types. With regard to the initialisation of storage, a function declared in *Relief* has a default definition so that initially its application gives only the undefined value. The extent of a non-lexical type, accessed by the *All* metafunction, is initially the empty set. Type declarations however are commands and are not referentially transparent because they alter the database schema. Since they contain no subexpressions and cannot appear as subexpressions they are omitted from the effects checking.

With regard to updates, user-defined functions are updated by the *def* and *del* functions which define and delete equations in the definition of a function. The extents of non-lexical types are altered by the *create* and *destroy* functions. An expression which directly or indirectly calls any of these four functions has the (W) property. An expression which directly or indirectly applies a user-defined function or examines the extent of a non-lexical type has a value which is dependent on the state of the system. Such an expression has the (R) property. Since no expressions in *Relief* can have the (A) property it is sufficient to classify them as belonging to one of the following three effect classes:

{W } { W,R }	- PROCEDURE class
{ R }	- OBSERVER class
{ }	- PURE class

Because *Relief* is a single-user sequential system we can say that an expression is referentially transparent if it makes no change to mutable storage under any condition. Thus an expression with the effect class OBSERVER or PURE is referentially transparent and has a value which is independent of the evaluation order of its subexpressions. The operational semantics of *Relief* allows the static scoping of referential transparency. Thus we arrive at the definition for the scope of referential transparency for an expression in *Relief*:

"The scope of an expression's referential transparency in Relief is the maximal enclosing expression which has no (W) effects."

In a multi-user system, PURE and OBSERVER expressions can still be treated as referentially transparent if there is a locking mechanism to prevent expressions with effect class PROCEDURE from changing the function and entity types on which they depend.

Our approach differs from that of Gifford and Lucassen in that we regard the effect class of an expression as a property orthogonal to its type. We regard their approach of annotating each function type with its effect class as unnecessarily complex and giving no practical benefit. In our implementation an expression's effect class is inferred from its subexpressions in tandem with its type. Thus whilst performed in parallel effects checking and type checking are conceptually independent.

In *Relief* the programmer declares the effect class of a function from the outset and the compiler ensures that the declaration is not violated. The effect class of a user-defined function must be either PROCEDURE or OBSERVER since it is an observer of its own mutable definition and hence possesses the (R) property. A user-defined function defaults to the effect class of OBSERVER, whereas an effect class of

PROCEDURE is specified by preceding the function name with "proc". The following examples illustrate this. Firstly declaring a PROCEDURE whose application conditionally creates an author and defines their surname and initials:

```
proc create_author : string string -> update;
def create_author sur init =>
    if [ a | for a in rev_surname sur;
                              initials a == init ] == [ ]
    then { a = create author; (def surname a => sur);
                        (def initials a => init) } fi;
```

Secondly declaring an OBSERVER function which is dependent on other functions. This example defines a person's salary as the sum of their basic pay and their bonus.

```
salary : person -> integer;
def salary x => basic x + bonus x;
```

Finally declaring an OBSERVER function which is only dependent on its own definition. This example extracts the second component of a tuple.

```
sec_comp : (alpha1 ** alpha2) -> alpha2;
def sec_comp (x,y) => y;
```

The effects checker ensures that the declarations are not violated by checking that the effect class of each defining equation is less than or equal to the declared effect class of the function. The inference rules used in the effects checker are given by the pseudocode below. The effect class of a pattern is PURE and since this is the least value it can be ignored where some other expression is present.

The fundamental inference rule is the one for application. The effect class of an application is the maximum of the effect class of the function and the effect class of its argument. This means that effect inference is pessimistic since it assumes that if an expression contains a side-effect then that side-effect will take place. This is of course not necessarily true with lazy evaluation. We now give the algorithm for the function *effect_chk* which infers the effect class of an expression. The declared effect class of a user-defined function is retrieved by calling the function *dec_effect*. Thus the value of "dec_effect(create_author)" is PROCEDURE assuming the declaration of create_author above.

```
effect_chk(E)    // effects checker
{
switch(E)
    case (E1 E2)      // application
    case ( e_let p = E2 in E1 ) //eager let expression
    case ( l_let p = E2 in E1 ) //lazy let expression
         e1 = effect_chk E1
         e2   = effect_chk E2
```

```
      return (max(e1 , e2) )
case (\p. E)// lambda abstraction
      return ( effect_chk E)
case Qual      //Qualifier in a list comprehension
      case ( for p in E)// generator
      case E             // filter
            return ( effect_chk E)
case VAR v  // a bound formal parameter, not
            // updateable, hence PURE
case NONLEX n      // a nonlexical entity
case CONFUN c      // a data constructor
case ETYPE         // a type as a parameter
case constant c    // eg integer or string
      return (PURE)
case FUN f         // a persistent function
      return(dec_effect(f))
case INFUN f       // a built-in function
      switch(f)
      case def f a1 .. an => rhs
            e = effect_chk (rhs)
            if e > dec_effect(f) then FAIL
      return(PROCEDURE)
      case del f a1 .. an
      case create t
      case destroy t
            return(PROCEDURE)
      OTHERWISE
            return(PURE)
```

This algorithm can be modified to prohibit subexpressions with the effect class PROCEDURE from parts of the language. We suggest that this restriction might be applied to list comprehension qualifiers so that optimisations such as those published by Trinder [Tri91] can be automatically performed.

8 Related Work

Ghelli et al. [GOPT92] have proposed an extension to list comprehension notation which provides update facilities for an object-oriented database language. The extension is achieved by adding a *do* qualifier which can perform actions on objects identified by earlier bindings in the list comprehension. They show that known optimisation techniques can be modified to operate in the presence of do qualifiers. Their technique however relies on the programmer restricting the use of side-effecting expressions to do qualifiers, there being no compile-time checking.

We briefly discuss three other approaches which promise efficient, functionally pure database update: linear types, monads and mutable abstract data types (MADTs).

They are all efficient since they can guarantee a single-threaded store so that updates can be done in-place.

8.1 Linear Types

A linear object is an unshared unaliased singly-referenced object and thus linear objects have the property that there is only one access path to them at any given time. A linear variable is a 'use-once' variable which can only be bound to a linear object and must be dynamically referenced exactly once within its scope. Linear type checking systems have been designed which can statically check that linear variables are neither duplicated nor discarded. These systems have a theoretical foundation in linear logic whence they derive their name.

The motivation for having linear types in functional languages is to capture the notion of a resource that cannot or should not be shared, hence the linear variable. A linear type system does however allow for the explicit creation, copying and destruction of linear objects. Whenever a program creates a linear object it must also be responsible for its destruction. One advantage of this is that there is no need for garbage collection in a purely linear language. For functional programmers linear types offer a way, amongst other things, of guaranteeing safe in-place updates since a linear object can have only a single reference.

Purely linear languages however are very restrictive since functions must always pass-on their linear arguments, explicitly destroy them or return them even when they are not changed. The programs in these languages tend to be somewhat difficult to read due to their heavy use of multiple returned values. Another problem with languages which support linear types is that they must also deal with nonlinear, i.e. shared, objects. This requires them to have a dual type checking system whereby most of the type inference rules have linear and nonlinear counterparts. The programmer must be aware of the distinction between linear and nonlinear values.

In the database context we want to treat database objects as linear when updating them but also to treat the same objects as shared when querying. However an object cannot be both linear and nonlinear at the same time. An interesting application of linear types to functional databases has been implemented by Sutton and Small [SS95] in their further development of PFL, a functional database language which uses a form of relation, called a selector, as its bulk data type. In the first version of PFL the incremental update of relations was achieved by means of the side-effecting functions *include* and *exclude*. A linear type checking system was introduced into PFL later on by changing these functions so that they took a linear relation name as an argument and returned the same relation name as a result. The type checking system worked by preventing the duplication or discarding of these relation names. Sutton and Small give examples of strict updating functions which show that the language is still update-complete within the constraints of this type checking. In order to reduce the number of parameters returned when querying they also allow a relation to be referred to by a nonlinear name. However a relation's linear name and nonlinear name cannot both appear within the same expression.

8.2 Monads

In-place update can also be guaranteed by wrapping up the state in a higher-order type which is accessed only by functions with the single-threaded property. The functions must support the creation, access and updating of the state within the type. Wadler has shown how monads can be used for this purpose [Wad95]. Monads, a category-theoretic notion, are defined by a triple: the type constructor M, and the operations *unit* and *bind*. Monadic programming is an implicit environment based approach to programming actions such as updating state and input/output. Actions on the environment have a special type which indicates what happens when the action is performed. The programmer composes actions with the two combinators unit and bind.

The monadic approach to handling state has both advantages and disadvantages when compared with the use of linear types. Its chief advantage is that it does not require a special type system beyond the usual Hindley-Milner one. Also because monads handle state implicitly it is argued that this can make an underlying algorithm more apparent since there is less parameter passing. Their main disadvantage is that they require a centralised definition of state. Although this restriction might initially appear to fit nicely with the use of a single conceptual data model in databases, in practice however this lack of granularity would have performance and concurrency implications. A treatment of multiple stores with different types, such as relations in a relational database, requires the definition of many monads and monad morphisms in order to support operations involving the different types. It may also be that for databases a two state monad is more appropriate in order to represent the current state and the state at the start of the transaction. This could then support the rollback operation. We consider these to be areas for further research.

Monads can also be defined in terms of *unit*, *map* and *join* operations which are a generalisation of the same operations which give the semantics of list comprehensions. Wadler has shown how the list comprehension notation can also be used to manipulate state monads [Wad90], each qualifier becoming an action on the state. The resulting programs however look very procedural.

8.3 Mutable Abstract Data Types

Monads in themselves do not enforce linearity on the encapsulated state, their operations must be carefully designed in order to ensure this. The use of abstract data types which encapsulate state and linear access to it has been investigated by Hudak [CH97]. A *mutable abstract datatype* or MADT is any ADT whose rewrite semantics permit "destructive re-use" of one or more of its arguments while still retaining confluence in a general rewrite system. A MADT is an ADT whose axiomatisation possesses certain linearity properties. Hudak shows how a number of MADTs can be automatically derived from such an ADT. Using an array as an example Hudak gives three ways of defining an appropriate MADT. These correspond to direct-style semantics, continuation passing style (CPS) and monadic semantics. A MADT is defined in terms of functions which generate, mutate and select state of a simple type.

Not only is the state hidden in a MADT but the linearity is too and so there is no need for the programs using MADTs to have a linear type system. It has been noted that programming in MADTs tends towards a continuation passing style no matter which form of MADT is used. MADTs are excellent at handling simple state but further research is required into handling bulk data types efficiently and into combining MADTs. Unlike monads there is no rigorous way of reasoning with a number of MADTs.

9 Conclusions

Whilst there is much promising research into handling state within functional languages the update problem in functional databases such as FDL remains problemmatical because the database is comprised of function definitions which must be both simple to apply and simple to update. The design of an uncomplicated integrated language to handle both queries and updates in an efficient purely functional manner is difficult. Whilst linear types could have been used it is our view that the result would have been less satisfactory and less programmer-friendly than the approach we have taken. The introduction of the four update functions described in section 4, whilst not functionally pure, has kept the language strongly data-model oriented. Also none of the approaches outlined in the section on related work have been used to address transaction handling which is a fundamental distinction between database programming and persistent programming. In contrast *Relief* does have a programmable transaction handling mechanism although this is not described here. (See [Mer98] for details of this).

Relief demonstrates that a functional interpreter based on graph reduction can be used to perform updates and that effects checking can scope referential transparency so that the usual functional optimisations can be used for queries and read-only subexpressions. The approach taken has allowed the benefits of lazy evaluation and static type checking to be retained. Updates are performed efficiently and the programmer does not have to learn a new type system. Whilst the eager let helps to make the sequencing of updates more explicit it is still necessary to understand the operational semantics of the pattern matcher when function arguments contain side-effects. Applicative order reduction would simplify these semantics but the lazy stream model used for file reading and the other advantages of lazy evaluation would be lost. Update by side-effect and effects checking are applicable to database systems supporting other data models.

Further work will centre on integrating *Relief* with the data model of *Hydra* [AK94] which distinguishes between navigable functions and non-navigable functions in a functional database. This may require the definition of further effect classes.

Acknowledgements: We would like to thank the referees and Alex Poulovassilis for their comments during the preparation of this paper. The first author was supported by the EPSRC and IBM UK Labs Hursley during this work.

10 References

[AK94] R. Ayres and P.J.H. King: Extending the semantic power of functional database query languages with associational facilities. In Actes du Xieme Congres INFORSID, pp301-320, Aix-en-Provence, France, May 1994.

[CH97] Chih-Ping Chen and Paul Hudak: Rolling Your Own Mutable ADT – A connection between Linear Types and Monads. ACM Symposium on Principles of Programming Languages, January 1997

[Der89] M. Derakhshan: A Development of the Grid File for the Storage of Binary Relations Ph.D. Thesis, Birkbeck College, University of London, 1989

[FH88] Anthony J. Field, Peter G. Harrison: Functional Programming. Addison-Wesley 1988

[GL86] David K. Gifford and John M. Lucassen: Integrating Functional and Imperative Programming. In Proceedings of the ACM Conference on Lisp and Functional Programming, Cambridge, Massachussets, pp28-39, ACM, 1986

[GOPT92] Giorgio Ghelli, Renzo Orsini, Alvaro Pereira Paz, Phil Trinder: Design of an Integrated Query and Manipulation Notation for Database Languages, Technical Report FIDE/92/41, University of Glasgow, UK, 1992.

[Hud89] Paul Hudak: Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, Vol. 21, No. 3, September 1989 pp359-411

[KDPS90] Peter King, Mir Derakhshan, Alexandra Poulovassilis, Carol Small: TriStarp - An Investigation into the Implementation and Exploitation of Binary Relational Storage Structures. Proceedings of the 8th BNCOD, York 1990

[Mer98] P.F.Meredith: Extending a Lazy Functional Database Language with Updates. Thesis for Submission. Birkbeck College, University of London, 1998

[NHS84] J. Nievergelt, H. Hinterberger, K.C. Sevcik: The Grid File: An Adaptable, Symmetric, Multikey File Structure. ACM TODS, Vol 9, No 1, March 1984, pp38-71

[Pou88] A. Poulovassillis: FDL: An Integration of the Functional Data Model and the Functional Computational Model, Proceedings of the 6th BNCOD, July 1988, Cambridge University Press, pp215-236.

[Pou92] A. Poulovassilis: The Implementation of FDL, a Functional Database Language. The Computer Journal, Vol 35, No 2, 1992

[PK90] A. Poulovassilis and P. J. H. King: Extending the Functional Data Model to Computational Completeness. Proceedings of EDBT'90, pp75-91, Venice 1990. Springer-Verlag, LNCS 416.

[Shi81] David W. Shipman: The Functionanal Data Model and the Data Language DAPLEX. ACM TODS, Vol 6, No 1, March 1981, pp140-173

[SS95] David Sutton, Carol Small: Extending Functional Database Languages to Update Completeness. Proceedings of 13th BNCOD, Manchester, 1995

[Tri91] Phil Trinder: Comprehensions, a Query Notation for DBPLs. The 3rd International Workshop on DBPLs, "Bulk Types and Persistent Data". August 1991, Nafplion, Greece. Morgan Kaufman Publishers.

[VV82] G.M.A. Verheijen, J. Van Bekkum: NIAM: An Information Analysis Method. In "Information Systems Design Methodologies: A Comparative Review", T.W.Olle et al. (eds), North-Holland, 1982

[Wad90] Philip Wadler: Comprehending Monads. ACM Conference on Lisp and Functional Programming, Nice, June 1990

[Wad95] Philip Wadler: Monads for Functional Programming. In "Advanced Functional Programming", Proceedings of the Bastad Spring School, May 1995, LNCS vol 925

Extending the ODMG Architecture with a Deductive Object Query Language

Norman W. Paton and Pedro R. Falcone Sampaio

Department of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, United Kingdom
Fax: +44 161 275 6236 Phone: +44 161 275 6124
E-mail: [norm,sampaiop]@cs.man.ac.uk

Abstract. Deductive database languages have often evolved with little regard for ongoing developments in other parts of the database community. This tendency has also been prevalent in deductive object-oriented database (DOOD) research, where it is often difficult to relate proposals to the emerging standards for object-oriented or object-relational databases. This paper seeks to buck the trend by indicating how deductive languages can be integrated into the ODMG standard, and makes a proposal for a deductive component in the ODMG context. The deductive component, which is called DOQL, is designed to conform to the main principles of ODMG compliant languages, providing a powerful complementary mechanism for querying, view definition and application development in ODMG databases.

1 Introduction

To date, deductive database systems have not been a commercial success. The hypothesis behind this paper is that this lack of success has been encouraged by the tendency from developers of deductive databases to build systems that are isolated from the widely accepted database development environments. The fact that deductive database systems tend to be complete, free-standing systems, makes their exploitation by existing database users problematic:

1. Existing data is not there. Existing databases in relational or object-oriented systems cannot be accessed by deductive databases without transferring the data into the deductive database system.
2. Existing tools are not there. Moving to a new database system means learning a new toolset, and coping with the fact that few deductive database systems have benefited from the levels of investment that have been directed towards leading relational or object-oriented products.
3. Existing programs are not there. Software that has been developed for non deductive databases has no straightforward porting route to a deductive environment.
4. Existing experience is not useful. There are very few developers for whom a deductive database would be a familiar environment.

These difficulties that face any developer considering using deductive databases with existing applications mean that even where there is an application that could benefit from a deductive approach, such an approach may well not be exploited. With new applications, points (2) and (4) remain as a disincentive, and the decision rests on the widely rehearsed debate on the relative strengths and weaknesses of deductive and other database paradigms. Here we note only that the decision to use a deductive database seems to be made rather rarely. In practice, the limited range and availability of deductive database systems will often lead to an alternative approach being used by default.

In the light of the above, we believe that it is not generally viable to make a case for deductive database systems as a replacement for existing approaches. There are sure to be a number of domains for which a deductive approach is very appropriate, but it is difficult to envisage widespread uptake of 'pure' deductive database systems.

The argument that deductive languages can provide a mechanism for querying, recursive view definition and application development that is *complementary* to existing database languages is much easier to make. Where deductive features are made available within an existing database system, the user can elect to exploit the deductive mechanism or not, depending on the needs of the application. Where the deductive mechanism is closely integrated with other language facilities, multi-paradigm development is supported, and thus the deductive language can be used in niche parts of an application if more widespread exploitation is not felt to be appropriate.

The argument that deductive database languages can be complementary to those of other paradigms is not new [17], but it is not only the principle of integration, but also the practice that counts. For example, [16] describe the integration of the deductive relational database Coral with C++, but the impedance mismatches that result are extreme, and the approach can be considered more of a coupling than an integration. In ROCK & ROLL [2] the integration is much more seamless, but neither the languages nor the data model used predated the development of the ROCK & ROLL system.

The argument for deductive languages in any form can be made on the basis of application experience [10], or on the grounds that an alternative style and more powerful declarative language extends the options available to programmers. Although the case that recursion is an important motivation for including deduction in databases has been used previously to the point of tedium, we note that ODMG languages do not currently provide any clean way of describing recursive relationships. As a simple example, the implementation of the simple ancestor relationship as a method using the ODMG C++ binding is presented in figure 1, for the database in figure 3. The code would be more complex if cycle detection was required.

In what follows this paper considers *how* a deductive language can be integrated into the ODMG architecture. The decision space that relates to the incorporation of deductive facilities into the ODMG model is presented in section 2. The deductive language DOQL is described in section 3, and its integra-

```
d_Set<d_Ref< Person >> * Person::ancestors const ()
{    d_Set<d_Ref<Person>> *the_ancestors = new d_Set<d_Ref<Person>>;
     //---- ancestors = parents Union ancestors (parents)
     if (father != 0) {
        the_ancestors->insert_element(father);
        d_Set<d_Ref<Person>> *f_grand_parents = father->ancestors();
        the_ancestors->union_with(*f_grand_parents);
        delete (f_grand_parents);}
      if (mother != 0)
      {  d_Set<d_Ref<Person>> *m_grand_parents = mother->ancestors();
         the_ancestors->insert_element(mother);
         the_ancestors->union_with(*m_grand_parents);
         delete (m_grand_parents);}
      return the_ancestors;
}
```

Fig. 1. C++ implementation of *ancestors()* method

tion into the ODMG environment is discussed in section 4. How DOQL relates to other languages for object-oriented databases is outlined in section 5, and a summary is presented in section 6. Throughout the paper, it is assumed that readers have some familiarity with the ODMG standard, and with deductive database technologies, as described in [4] and [5], respectively.

2 The Design Space

This section considers the issues that must be addressed when seeking to add deductive facilities into the ODMG standard. The fundamental questions are *what deductive facilities should be supported* and *what relationship should the deductive facilities have to other components in an ODMG database.*

2.1 The Architecture

The introduction of deductive facilities into the ODMG architecture is made straightforward by the presence of OQL as a role model. The inclusion of the deductive language in a way that mirrors that of OQL minimizes the amount of infrastructure that existing OQL users must become acquainted with to exploit the deductive language. The following architectural principles can be followed in the development of the deductive extension.

1. Existing components should not be modified, and extensions should be minimal.
2. The deductive component should make minimal assumptions about the facilities of the ODMG system with which it is being used, and should be amenable to porting between ODMG compliant systems.

The consequences of the first of these principles can be considered in the context of the existing ODMG components as follows:

- *The data model and ODL:* the inclusion of a deductive language should necessitate no changes to the structural data model, but will require some extensions to operation definition facilities, as deductive rules do not have pre-defined input and output parameters.
- *OQL:* the presence of an existing declarative language raises the question as to whether or not one should be able to make calls to the other. There is no doubt that the ability to make calls from OQL to the deductive language (and potentially the reverse) could be useful, but such extensions would impact on principle (2) above.
- *Language bindings:* the existing language bindings need not be changed to accommodate a new embedded language. The deductive language should be callable from C++, Java and Smalltalk in a manner that is reminiscent of the embedding of OQL, through the addition of new library calls for this purpose.

The consequences of the second of the principles listed above depend on whether or not the developer has access to the source code of an ODMG compliant system. The assumption that is being made (because it is true for us) is that the deductive language is being implemented without access to the source of any ODMG compliant system. This has the following consequences for the implementation:

- *Query language optimization/evaluation:* this has to be written from scratch, as reusing the existing infrastructure developed for OQL is only an option for vendors. Implementing the optimiser and evaluator specifically for the deductive language also eases porting of the deductive component.
- *Data model interface:* it is necessary to have access to a low level API for the underlying database (such as O_2API), as the deductive system often needs to perform operations on objects for which the types are not known at system compile time.

The architecture adopted for DOQL is illustrated in figure 2. The DOQL compiler and evaluator is a class library that stores and accesses rules from the ODMG database – the rule base is itself represented as database objects. The class library is linked with application programs and the interactive DOQL interface. The interactive interface is itself a form of application program.

2.2 The Language

Research on deductive languages for object databases has been underway for around 10 years now, and a considerable number of proposals have been made (see [15] for a survey covering 15 proposals). These proposals differ significantly in the range of facilities supported by the logic language, so that a spectrum can be identified in which ROLL [2] is perhaps the least powerful and ROL [12]

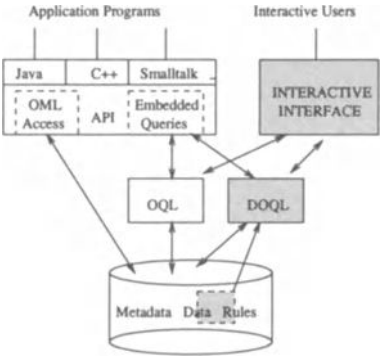


Fig. 2. Architecture diagram showing location of DOQL compiler/evaluator.

perhaps the most comprehensive. In practice, the range of facilities supported depends to a significant extent on the context. For example, ROLL is deliberately left simple because it is closely linked to the imperative language ROCK, whereas ROL is a single-language system with more limited access to complementary, non-deductive facilities. It is thus important to identify some principles that can be used to guide the development of an ODMG compliant DOOD. We suggest that an ODMG compliant deductive language should:

1. Be able to access all ODMG data types and should only be able to construct values that conform to valid ODMG data types.
2. Be usable to define methods that support overloading, overriding and late binding.
3. Be able to express any query that can be written in OQL.
4. Be usable embedded or free standing.

It is straightforward to think up additional principles (e.g. the deductive language should be usable to express integrity constraints; the deductive language should be able to update the underlying database; the deductive language should extend the syntax of OQL), but the more are added, the more constraints/burdens are placed on developers in terms of design and implementation. DOQL supports the 4 principles enumerated above, but none of those stated in this paragraph.

In general terms, the principles stated above indicate that the deductive language should be tightly integrated with the model that is at the heart of the ODMG standard, should support object-oriented programs as well as data, should extend what is already provided in terms of declarative languages, and should not be bound tightly to a specific user environment. The principles also avoid a commitment to areas of deductive language design that raise open issues in terms of semantics or the provision of efficient implementations. The language described in section 3 contains only language constructs that have been supported elsewhere before, and thus can be seen as a mainstream deductive

database language. The novelty of the approach is thus not in the individual language constructs, but in the context within which it fits.

3 The DOQL Language

This section describes the DOQL language, enumerating its constructs and illustrating them using an example application.

3.1 Example Application

The example application, from [8], describes employees and their roles within projects and companies. Figure 3 presents the structure of the database using UML [14].

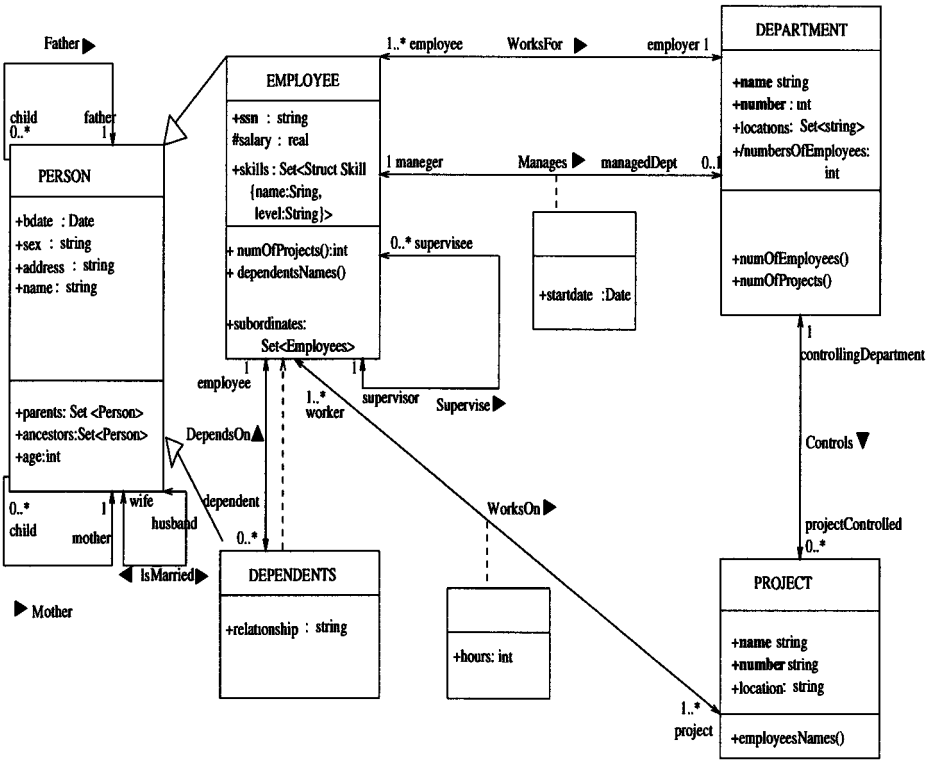


Fig. 3. Example Application

3.2 Queries

Two different approaches were considered in the definition of the syntax of the deductive language:

- Extend OQL to support recursion.
- Define a rule-based query language tailored to provide reasoning over the ODMG data model.

The main attraction of the first approach is in the possibility of reusing the existing OQL syntax. This advantage, though, does not supplant the difficulties involved in providing comprehensive deductive definitions in the straightjacket of the **Select From Where** syntax block[1]. DOQL thus adopts a syntax that is familiar from other DOOD proposals, but adapted for use in the ODMG model.

DOQL is a typed logic-based rule language supporting declarative queries and declarative rules for intensional definitions. The rule language follows the Datalog rule style, but combines the positional and named approaches to attribute denotation and extends Datalog's atoms and terms to deal with the properties of the underlying ODMG model. Variables range over instances of types declared in an ODMG schema. The language is composed of two main syntactic categories: *terms* and *literals*. Non negated literals are also called *atoms*.

Terms denote elements in the domain of interpretation and define the alphabet of the deductive language. Terms can be variables, atomic or complex literals and evaluable expressions:

- *Variables:* represented by alphanumeric sequences of characters beginning with an uppercase letter (e.g. `X`, `Names`).
- *Atomic Constants:* representing ODMG atomic literals (e.g., `2.32`, `'Alan Turing'`, `13`, `true`, `nil`).
- *Compound Constants:* representing structured or collection literals defined in the ODMG model. (e.g., `struct(name:'programming',level: 'high')`, `set(1,2,3,5)`, `bag(1,1,3,3)`, `list(1,1,2,4)`).
- *DB entry names:* representing the set of names associated with root objects and extents in an ODMG schema.
- *Expressions:* evaluable functions that return an ODMG structural element (literal or object). Each expression has a type derived from the structure of the query expression, the schema type declarations or the types of named objects. Expressions can have other terms as parameters and are divided in terms of the following categories:
  path expressions: traversal paths starting with a variable or a DB entry name, that can span over attributes, operations and relationships defined in type signatures of an ODMG schema (e.g.,`Z.age`, `john.subordinates`). Path expressions traversing single valued features in type signatures can be composed forming paths of arbitrary lengths (e.g., `john.dept.name`).

[1] It has recently been proposed that the inclusion of non-linear recursion in SQL be delayed until SQL4 [9].

aggregates and quantified expressions: predefined functions applied to terms denoting collections. Complex expressions can be formed by applying one of the operators (`avg`, `sum`, `min`, `max`, `count`, `exists`, `for-all`) to pairs *(T:F)*, where T is a term and F is a formula formed by a conjunction of literals. T and F can share common variables, and other variables can appear in F. Every variable that appears in T must also appear in F and cannot appear elsewhere outside the scope of the operator. The following expressions show the use of aggregates and quantifiers:

```
count(X.dependents), max(X.age: persons(X))
exists(X: persons(X), X.age > 110)
for-all(X: employees(X), X.age >= 18)
```

arithmetic expressions: (A + B), (A * B), (A / C), (A - C).
string concatenation: (A || B).
set expressions: (A `union` B), (A `intersect` B), (A `except` B).

Literals denote atomic formulas or negated atomic formulas that define propositions about individuals in the database. The types of atomic formulas are:

- *collection formulas*: enable ranging over elements of collections. A collection can be a *virtual collection* defined by a previous rule or a *stored collection* defined in the extensional database. Molecular groupings of collection properties can be done using the operator (`[ ]`). The operator (`=>`) can be used to finish path expressions that end in a collection, resulting in collection formulas. The following formulae show the diversity of possibilities in ranging over collections, depending on the needs of each rule. In the first formula, the identities of the collection instances represent the focus point. In the second formula, the items needed are the values for the properties `name` and `age` of each collection instance. The third formula binds variables to identities and properties of the elements in the collection. The fourth formula denotes all the subordinates of `john` and the last formula ranges over a collection of structured literals that represent information about `john`'s skills. Collections of literals can only use the molecular notation due to the absence of object identity in their elements.

```
persons(X)
persons[name=N, age=Y]
persons(X)[name=N, age=Y]
john.subordinates=>X
john.skills=>[name=N, level=L]
```

- *element formulas*: enable ranging over (1:1) relationships and object-valued attributes (without inverses) in the extensional database. The following formulae are examples of element formulas. In the first formula, `john` is a name (DB entry point) in the schema.

```
john.spouse->X
L.spouse->X
```

- *operation formulas*: denote formulas of type <*Term*> Operator <*Term*> or just <*Term*>, where the latter is a boolean expression. Operation formulas encompass relations between elements and collections (`in`), strings (`substr`, `like`, `lower`), numbers (<=, >=, <, >) and also the notions of value equality (=, <>) and OID equality (==, !=).

```
X.age <= 90
exists(X: employees(X), struct(name:'coding',level:'high') in X.skills)
john.father == mary.father
```

- *method calls*: denoted by the syntactic form:

  $< method\_name >(arg_1, ...arg_n)[- >< result >]$
  $< method\_name >(arg_1, ...arg_n)[=>< result >]$
  $< method\_name >(arg_1, ...arg_n)[=< result >]$

  This allows user-defined ODMG operations to be invoked from DOQL. The three forms return individual elements (objects or structured literals), collections, and simple values, respectively. The invoked operations should be side-effect free, a criterion that cannot be guaranteed or tested for in existing ODMG environments. It is thus up to the user to exercise caution in embedding method invocations, and the system should support an option of forbidding method calls from within DOQL (for example, through an environment variable). It is not possible to call methods in which a single argument position is used for both input and output, as this would require assignment to a bound DOQL variable.
- *DOQL goals*: these are described below, along with consideration of DOQL rule and method definitions.

3.3 DOQL Methods and Rules

Rules are used to implement deductive methods and define virtual collections over ODMG databases. A rule is a clause of the form: $H \leftarrow L_1, L_2, L_3, \ldots L_n$, where H is the *Head* of the rule and the body is a sequence of literals denoting a formula. The syntactic form of the head depends on whether the rule is a *regular clause* or a *method clause.*

Regular Clauses In regular clauses, the head of the rule is of the form:

$< rulename > (arg_1, ..., arg_n)$

where each arg_i is a variable, an atomic constant, a compound constant, or a grouping expression applied to a term. For example, in the following rule, `D` is `local` if it is a department that controls only local projects:

```
local(D) :- departments(D),
            for-all(Projects: D.projectControlled=>Projects,
                       Projects.location='local').
```

Rules can be recursive, and can be invoked from queries or rules:

```
parent(X,Y) :- persons(X), X.father->Y.
parent(X,Y) :- persons(X), X.mother->Y.
relative(X,Y) :- parent(X,Y).
relative(X,Y) :- parent(X,Z), relative(Z,Y).
```

Grouping is the process of grouping elements into a collection by defining properties that must be satisfied by the elements. Grouping is restricted to rules that contain only a single clause and to a single argument position in the head of the clause. The grouping construct is expressed by the symbols { Var_Name } for set groupings, < Var_Name > to group elements as bags and [Var_Name | $a_i : o_i, \ldots a_n : o_n$] to group as lists, sorting the list according to the attributes $a_i, \ldots, a_n$, each attribute defining a letter (a=ascending, d=descending) for the ordering field o_i that defines the order of the elements of the list according to that attribute. The following example shows the grouping of the relatives of a person as a list sorted in ascending order of the age attribute.

```
relatives_of(X,[Y|age:a]) :- relative(X,Y).
```

Method Clauses In method clauses, the head of the rule is of the form:

$< recipient >::< method - name > (arg_1, ..., arg_n)$

where *recipient* is a variable, and arg_i are as for regular clauses. For example, the following method rule reinterprets `relative` as a deductive method on `person`:

```
extend interface PERSON {
    P::relative(R) :- parent(P,R).
    P::relative(R) :- parent(P,Z), Z::relative(R).
}
```

The approach to overriding and late binding follows that of ROCK & ROLL [7]. In essence, methods can be overridden, and the definition that is used is the most specialised one that is defined for the object that is fulfilling the role of the message recipient.

3.4 Stratification, Safety and Restrictions

DOQL adopts conventional restrictions on rule construction to guarantee that queries have finite and deterministic results. In particular: each variable appearing in a rule head must also appear in a positive literal in the rule body; each variable occurring as argument of a built-in predicate or a user-defined ODMG operation must also occur in an ordinary predicate in the same rule body or must be bound by an equality (or a sequence of equalities) to a variable of such an ordinary predicate or to a constant; all rules are stratified; a rule head can have at most one grouping expression; the type of each argument of an overridden DOQL method must be the same as the type of the corresponding argument in the overriding method; all rules are statically type checked, using a type inference system.

4 DOQL In Context

This section describes how DOQL fits into the ODMG environment, showing how rule bases are created and managed, how rules are invoked, and how DOQL is incorporated into imperative language bindings.

4.1 Managing Rule Bases

Rules can be created either transiently, or in rule bases that are stored persistently in the ODMG compliant database. Rule bases are created using the `create_rulebase` operation, which takes as parameter the name of the rulebase to be created. The inverse of `create_rulebase` is `delete_rulebase`. These commands can be run from the operating system command line, from the interactive DOQL system, or from executing programs.

Once a rulebase has been created, rules can be added to it using the `extend rulebase` command, which can also be run from the operating system command line, from the interactive DOQL system, or from executing programs. For example, figure 4 extends the rulebase `employment` with both regular and method rules.

```
extend rulebase employment {
// Extend EMPLOYEE with an operation related_subordinate
// that associates the EMPLOYEE with subordinates who are related to them
 extend interface EMPLOYEE {
    Emp::related_subordinate(R) :- Emp::relative(R), Emp::subordinate(R).
    Emp::subordinate(S) :- Emp[supervisee=>S].
    Emp::subordinate(S) :- Emp[supervisee=>Int], Int::subordinate(S).
   }
 // A dodgy manager is one who is the boss of a relative
 dodgy_manager(M) :- employees(M), exists(S: M::related_subordinate(S)).
}
```

Fig. 4. Extending a rule base.

4.2 Rule Invocation

Queries are evaluated over the current state of the database and rule base, with different query formats depending on the nature of the query. Rules can only be invoked directly from within DOQL queries and rules. The different formats are:

- Single element queries (`SELECT ANY`): this form of query returns a single element from the set of elements that results from the evaluation of the goal. The choice of the element to be returned is non-deterministic.

- Set of elements queries (SELECT): this form of query returns all elements that satisfy the goal.
- Boolean queries (VERIFY): this form of query has a boolean type as query result. The result is the value yielded by the boolean formula given as a parameter.

The query formats can be summarized according to the following general expressions:

```
SELECT [ANY] Result_Specifier
    FROM    DOQL Query
    [WITH   Rule_Statements]
    [USING  Rulebase]

VERIFY Boolean_Formula
    [WITH   Rule_Statements]
    [USING  Rulebase]
```

The Result_Specifier is a comma separated list of arguments that are the same as the arguments allowed in a rule head, and Rulebase is a comma separated list of persistent rulebase names. For example, the following query associates each employee with a set of their related subordinates, using rules from the employment rulebase.

```
SELECT E,{S}
FROM   E::related_subordinate(S)
USING  employment
```

The verification format requires that the operand of the verify return a boolean value:

```
VERIFY exists(E:employees(E), dodgy_manager(E))
USING  employment
```

The WITH clause is used to allow the specification of rules that exist solely for the purpose of answering the query (i.e. which are not to be stored in the persistent rule base). For example, the following query retrieves the names of the employees who have more than 5 related subordinates:

```
SELECT Name
FROM   employees(E)[name=Name],
       num_related_subordinates(E,Number), Number > 5
WITH   num_related_subordinates(E,count({S})) :- E::related_subordinate(S)
USING  employment
```

The above query forms can be used either in the interactive DOQL interface or in embedded DOQL.

4.3 Embedded DOQL

DOQL programs can be embedded in host programming languages in a manner similar to that used for OQL [4]. DOQL statements are embedded as parameters of calls to the API function d_doql_execute (d_DOQL_Query Q, Type_Res

Result)[2]. The function receives two parameters. The first parameter is a query container object formed by the query form described in section 4.2 along with any input arguments to the query. The second parameter is the program variable that will receive the result of the query.

Figure 5 is a code fragment that shows the embedded form of DOQL for C++. First some variables are declared for the input parameter to the query (`pedro`) and the result (`the_ancestors`). Then the query object `q1` is created with the query as the parameter of the constructor function. It is then indicated that `pedro` is the (first and only) parameter of `q1` – this parameter is referred to within the text of the query as `$1`. Finally, the query is executed by `d_doql_execute`, and the results are placed in `the_ancestors`.

```
d_Ref <Person> pedro = ...;
d_Set < d_Ref <Person>> *the_ancestors = new Set(Person);
d_DOQL_Query q1("SELECT Y
                 FROM relative($1,Y)
                 WITH  relative(X,Y) :- persons(X), X.father->Y.
                       relative(X,Y) :- persons(X), X.mother->Y.
                       relative(X,Y) :- relative(X,Z), relative(Z,Y).");
q1 << pedro;
d_doql_execute(q1, the_ancestors);
```

Fig. 5. Example embedded DOQL code fragment

5 Related Work

This section outlines a range of features that can be used to compare the query components of proposals that integrate declarative query languages and imperative OO programming languages in the context of OODBs.

Bidirectionality of calls: relates to the flexibility of the integration approach. In unidirectional systems, one of the languages can call the other, but not vice-versa.

Restructuring: relates to the operations available in the declarative language that can be used to reorganize the queried data elements to yield more desirable output formats.

Type system: relates to the underlying type system used across the integrated languages.

[2] A general syntax is used to show concepts. The specific syntax varies for each language binding (e.g. for C++, template<class T> void d_doql_execute(d_DOQL_Query &query, T &result)).

Query power: relates to the query classes supported by the query language. (FOLQ = first-order queries, FIXP = fixpoint queries, HOQ = higher-order queries).

View mechanism: relates to the features provided by the query language for the definition of derived attributes, derived classes through filtering, derived relations through composition, and derived classes through composition applying OID invention.

Table 1 shows a comparison between the query facilities supported by DOQL and the query components of Chimera [6], Coral++ [16], OQL [4], OQLC++ [3], Noodle [13] and ROCK & ROLL [2].

Language	Criteria				
	Bidirection of Calls	Restructuring Operators	Type System	Query Power	View Mechanism
Chimera	no	unnesting	Chimera object model	FOLQ, FIXP	attributes, classes (filtering), relations
Coral++	no	set grouping, unnesting	C++	FOLQ, FIXP	relations
DOQL	yes	(set,bag,list) grouping, unnesting	ODMG object model	FOLQ, FIXP	classes (filtering), relations
Noodle	yes	(set,bag) grouping, unnesting	Sword object model	FOLQ, FIXP HOQ	relations
OQL	yes	(set,bag,list) grouping, unnesting	ODMG object model	FOLQ	classes (filtering), relations, classes (OID invention)
OQLC++	yes	unnesting	C++	FOLQ	no
ROCK & ROLL	yes	unnesting	Semantic object model	FOLQ, FIXP	no

Table 1. Query languages for object databases

Some important aspects of our design are:

- language integration is done using a standard object-model as the underlying data model of the deductive system.
- the approach to integration complies with the call level interface defined by the ODMG standard, reusing existing compiler technology for other ODMG compliant languages and providing portable deduction within the imperative languages without the need for changes in syntax or to the underlying DBMS.

- bidirectionality of calls between the integrated languages.
- powerful aggregate and grouping operators that can be used for restructuring data in applications that require summaries, classifications and data dredging.
- updating and control features are confined within the imperative language

Views in DOQL are considered subproducts of queries, and in particular, results of queries are typically associations (complex relations). This context differs from the semantics of views proposed in [11, 1] by not inventing OIDs, by not having view declarations as a part of the DB schema, and by not having methods attached to views obtained through composition.

Language integration proposals can also be compared based on the seamlessness of the language integration (see [2]) or based on the object-oriented and deductive capabilities supported (see [15]). Using the criteria of [2], DOQL can be seen to support evaluation strategy compatibility, (reasonable) type system uniformity and bidirectionality, but to lack type checker capability and syntactic consistency. The lack of type checker capability means that embedded DOQL programs can cause type errors at runtime – obtaining compile time type checking of embeddded DOQL would require changes to be made to the host language compiler. The lack of syntactic consistency is unavoidable, as DOQL can be embedded in any of a range of host languages, and thus cannot hope to have a syntax that is consistent with all of them.

6 Summary

This paper has proposed a deductive language for use in ODMG compliant database systems. The principal motivation for the development of such a language is the belief that failure to make deductive systems fit in with existing database models and systems has been a significant impediment to the widespread exploitation of deductive technologies in databases.

The language that has been proposed, DOQL, can be used both free-standing and embedded within the language bindings of the ODMG model. Rules can be defined in a flat clause base or as methods attached to object classes. The paper has described both the language and how it relates to other ODMG components both in terms of the implementation architecture and as a user language.

Acknowledgements: The second author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) – Grant 200372/96-3.

References

1. S. Abiteboul and A. Bonner. Objects and views. In *Proc. of the ACM-SIGMOD Int. Conference on Management of Data*, pages 238–247, 1991.
2. M. L. Barja, N. W. Paton, A. A. Fernandes, M. Howard Williams, and Andrew Dinn. An effective deductive object-oriented database through language integration. In *Proc. of the 20th VLDB Conference*, pages 463–474, 1994.

3. J. Blakeley. OQLC++: Extending C++ with an object query capability. In Won Kim, editor, *Modern Database Systems*, chapter 4, pages 69–88. Addison-Wesley, 1995.
4. R. Cattel and Douglas Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufman, 1997.
5. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990.
6. S. Ceri and R. Manthey. Consolidated specification of chimera (cm and cl). Technical Report IDEA.DE.2P.006.1, IDEA - ESPRIT project 6333, 1993.
7. A. Dinn, N. W. Paton, M. Howard Williams, A. A. Fernandes, and M. L. Barja. The implementation of a deductive query language over an OODB. In *Proc. 4th Int. Conference on Deductive and Object-Oriented Databases*, pages 143–160, 1995.
8. R. Elmasri and S. Navathe. *Fundamentals of Database Systems 2nd. Edition.* Addison-Wesley, 1994.
9. S. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in sql. Technical Report X3H2-96-075r1, International Organization for Standardization, 1996.
10. O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille. Applications of deductive object-oriented databases using del. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, chapter 1, pages 1–22. Kluwer Academic Publishers, 1995.
11. W. Kim and W. Kelley. On view support in object-oriented database systems. In W. Kim, editor, *Modern Database Systems*, chapter 6, pages 108–129. Addison-Wesley, 1995.
12. M. Liu. Rol: A deductive object base language. *Information Systems*, 21(5):431–457, 1996.
13. I. S. Mumick and K. A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proc. of the Third Intl. Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 360–378. Springer-Verlag, 1993.
14. Rational Software Corporation. *Unified Modeling Language 1.0 - Notation Guide*, 1997.
15. Pedro R. F. Sampaio and Norman W. Paton. Deductive object-oriented database systems: A survey. In *Proceedings of the 3rd International Workshop on Rules in Database Systems*, volume 1312 of *LNCS*, pages 1–19. Springer-Verlag, 1997.
16. D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 158–170, 1993.
17. J. Ullman and C. Zaniolo. Deductive databases: Achievements and future directions. *ACM - SIGMOD Records*, 19(4):75–82, December 1990.

The Chase of Datalog Programs

Nieves R. Brisaboa[1], Agustin González[2], Héctor J. Hernández[2], and José R. Paramá[1]

[1] Facultad de Informática. Univ. da Coruña. A Coruña. Spain.
{brisaboa,parama}@udc.es
[2] Laboratory for Logic, Databases, and Advanced Programming,
Dept. of Co. Science, NMSU, Las Cruces, NM, USA 88003-0001.
{hector,agonzale}@cs.nmsu.edu *** †

Abstract. The chase of datalog programs is a new way to reason about datalog programs that are evaluated on databases consistent with a set of constraints. It is an equivalence-preserving program transformation that uncovers properties of datalog programs that must hold when they are evaluated on consistent databases. Here, we summarizes our research.

1 Introduction

To understand the interaction of datalog programs with that class of databases we define the *chase* [2,3] in the context of deductive databases. Our chase outputs programs which are, in general, "simpler" than the input programs, since they will contain fewer distinct variables as a result of the equating of variables done by the chase.

Example 1. Let $P = \{r_0, r_1\}$, where:
$r_0 = p(X,Y) \ :- \ e(X,Y)$
$r_1 = p(X,Y) \ :- \ e(Z,X), e(X,Z), p(Z,Y)$

Let $F = \{f = e : \{1\} \rightarrow \{2\}\}$; f means that if we have two atoms $e(a,c)$ and $e(a,b)$ in a database consistent with F, then $c = b$. Let us denote the databases that are consistent with F by $SAT(F)$. Then chasing in the natural way [2,3] the trees for the expansions $r_1 \circ r_0$, $r_1 \circ r_1 \circ r_0$, $r_1 \circ r_1 \circ r_1 \circ r_0$, ..., that is, equating variables, as implied by the fd f, in atoms defined on e in those expansions, we can obtain the datalog program $P' = \{s_0, s_1, s_2\}$, the chase of P wrt F, where:
$s_0 = r_0$
$s_1 = p(X,X) : -e(Z,X), e(X,Z), p(Z,X)$
$s_2 = p(X,Z) : -e(Z,X), e(X,Z), p(Z,Z)$

*** H.J. Hernández is also affiliated with Inst. de Ing. y Tec., Univ. Autónoma de Cd. Juárez, Ch., México

† This work was partially supported by NSF grants HRD-9353271 and HRD-9628450, and by CICYT grant Tel96-1390-C02-02.

P and P' are equivalent on databases in $SAT(F)$. The idea behind the chase of a datalog program is to obtain an equivalent datalog program such that the recursive rules have a least number of different variables; variables in the original recursive rule(s) get equated accordingly to the fds given. In fact, because of the equating of variables, our chase transforms P into P', which is equivalent to a nonrecursive datalog program on databases in $SAT(F)$!

Thus the chase of datalog programs is a very powerful rewriting technique that let us uncover properties of datalog programs that are to be evaluated on consistent databases. In particular, we can use it to find a minimal program, one without redundant atoms or rules, that is equivalent to the given program over databases in $SAT(F)$. This extends Sagiv's results in [4] regarding containment, equivalence, and optimization of datalog programs that are evaluated on $SAT(F)$. We can use the chase of programs to uncover other properties, e.g., to find out whether a set of fds F implies an fd g on a program P [1]; F *implies* g on P if for any database d in $SAT(F)$, $P(d)$ satisfies g. The following example illustrates this.

Example 2. Assume that P'' consists of the rules $r_2 = p(X,Y) \ :- \ e(X,Y)$ and $r_3 = p(X,Y) \ :- \ e(X,Y), e(X,Z), p(Z,Y)$. Then it is difficult to see why $F = \{f = e : \{1\} \rightarrow \{2\}\}$ implies on P'' the fd $g = p : \{1\} \rightarrow \{2\}$. The chase of P'' wrt F helps us to see why.

Since the chase of P'' wrt F is the program $\{r_2, p(X,Z) \ :- \ e(X,Z), p(Z,Z)\}$, it is now easy to see that on any database the fixed point of p is exactly a copy of the facts defined on e — because the atoms defined on p and e in both rules have exactly the same arguments— and, thus, for any database d satisfying F, $P''(d)$ satisfies g. Therefore the chase of P'' allows us to see that F implies g on P''.

Using the chase of datalog programs, we prove that: (1) the FD-FD implication problem [1] is decidable for a class of linear datalog programs, provided that the fds satisfy a minimality condition; (2) the FD-FD implication problem can still be decided when we relax the linearity and minimality conditions on datalog programs of the previous class, provided that the set of recursive rules of the program preserves a set of fds derived from the input fds.

References

1. S. Abiteboul and R. Hull. Data Functions, Datalog and Negation. In *Porc. Seventh ACM SIGACT-SIGMOD-SIGART. Symposium on Principle of Database Systems,* pp 143-153, 1998.
2. A.V. Aho, C. Beer, and J.D. Ullman. The Theory of Joins in Relational Databases. *ACM-TODS*, 4(3) pp 297-314, 1979.
3. D. Maier, A.O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM-TODS*, 4(4) pp 455-469, 1979.
4. Y. Sagiv. Optimizing Datalog Programs. In J. Minker, editor *Foundations of Deductive Databases and Logic Programming*, chapter 17 pp 659-698. Morgan Kauffmann Publishers, 1987.

Describing and Querying Semistructured Data: Some Expressiveness Results

Natasha Alechina[1] and Maarten de Rijke[2]

[1] School of Computer Science, University of Birmingham
Birmingham, B15 2TT, England
N.Alechina@cs.bham.ac.uk
WWW home page: http://www.cs.bham.ac.uk/~nxa
[2] ILLC, University of Amsterdam, Pl. Muidergracht 24
1018 TV Amsterdam, The Netherlands
mdr@wins.uva.nl
WWW home page: http://www.wins.uva.nl/~mdr

Data in traditional relational and object-oriented databases is highly structured and subject to explicit schemas. Lots of data, for example on the world-wide web is only *semistructured.* There may be some regularities, but not all data need adhere to it, and the format itself may be subject to frequent change.

The important issues in the area of semistructured data are: how to describe (or constrain) semistructured data, and how to query it. It is generally agreed that the appropriate data model for semistructured data is an edge-labeled graph, but beyond that there are many competing proposals. Various constraint languages and query languages have been proposed, but what is lacking so far are 'sound theoretical foundations, possibly a logic in the style of relational calculus. So, there is a need for more works on calculi for semistructured data and algebraizations of these calculi' [Abiteboul 1997].

One of the main methodological points of this paper is the following. There are many areas in computer science and beyond in which describing and reasoning about finite graphs is a key issue. There exists a large body of work in areas such as feature structures (see, for example, [Rounds 1996]) or process algebra [Baeten, Weijland 1990,Milner 1989] which can be usefully applied in database theory. In particular, many results from modal logic are relevant here. The aim of the present contribution is to map new languages for semistructured data to well-studied formal languages and by doing so characterise their complexity and expressive power.

Using the above strategy, we study several languages proposed to express information about the format of semistructured data, namely data guides [Goldman, Widom 1997], graph schemas [Buneman et al. 1997] and some classes of path constraints [Abiteboul, Vianu 1997]. Among the results we have obtained are the following:

Theorem 1. *Every set of (graph) databases defined by a data guide is definable by an existential first-order formula.*

Theorem 2. *Every set of (graph) databases conforming to an acyclic graph schema is definable by a universal formula.*

Every set of (graph) databases conforming to an arbitrary graph schema is definable by a countable set of universal formulas from the restricted fragment of first-order logic.

We also argue that first-order logic with transitive closure FO(TC) introduced in [Immerman 1987] is a logical formalism which is best suited to model navigational query languages such as Lorel [Abiteboul et al. 1997] and UnQL [Buneman et al. 1997]. We give a translation from a Lorel-like language into FO(TC) and show that the image under translation is strictly less expressive than full first-order logic with binary transitive closure.

Theorem 3. *Every static navigational query is expressible in FO(TC).*

Theorem 4. *The image under translation into FO(TC) of static navigational queries is strictly weaker than FO(TC) with a single ternary predicate Edge and binary transitive closure.*

In our ongoing work, we are investigating the use of FO(TC) for query optimisation, and we are determining complexity characterisations of the relevant fragments of FO(TC), building on recent results by [Immerman, Vardi 1997] on the use of FO(TC) for model checking.

References

[Abiteboul 1997] Abiteboul, S.: Querying semi-structured data. Proc. ICDT'97 (1997)

[Abiteboul et al. 1997] Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. J. of Digital Libraries, **1** (1997) 68–88

[Abiteboul, Vianu 1997] Abiteboul, S., Vianu, V.: Regular path queries with constraints. Proc. PODS'97 (1997)

[Baeten, Weijland 1990] Baeten, J.C.M., Weijland, W.P.: Process Algebra. Tracts in Theoretical Computer Science, Vol. 18. Cambridge University Press (1990)

[Buneman et al. 1997] Buneman, P., Davidson, S., Fernandez, M., Suciu, D.: Adding structure to unstructured data. Proc. ICDT'97 (1997)

[Goldman, Widom 1997] Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. Proc. VLDB'97 (1997)

[Immerman 1987] Immerman, N.: Languages that capture complexity classes. SIAM J. of Comput. **16** (1987) 760 – 778

[Immerman, Vardi 1997] Immerman, N., Vardi, M.: Model Checking and Transitive Closure Logic. Proc. CAV'97 (1997) 291–302

[Milner 1989] Milner, R.: Communication and Concurrency. Prentic Hall (1989)

[Rounds 1996] Rounds, W.C.: Feature logics. In: van Benthem, J., ter Meulen, A. (eds.): Handbook of Logic and Language. Elsevier (1996)

The TENTACLE Database System as a Web Server

Marc Welz[1] and Peter Wood[2]

[1] University of Cape Town
Rondebosch, 7701 RSA
mwelz@cs.uct.ac.za
[2] King's College London
Strand, London, WC2R 2LS UK
ptw@dcs.kcl.ac.uk

Abstract. We describe the TENTACLE system, an extensible database system which uses a graph data model to store and manipulate poorly structured data. To enable the database to be tailored to particular problem domains, it has been equipped with a small embedded interpreter which can be used to construct the equivalent of customised views of or front-ends to a given semi-structured data domain. To demonstrate the capabilities of the system we have built a web server directly on top of the database system, making it possible to provide a clean mapping between the logical structure of the web pages and the underlying storage system.

1 Introduction

TENTACLE is an extensible database system which attempts to take a different approach to modelling, manipulating and presenting complex or poorly structured application domains. As data model we have chosen a graph-based approach. Graph models provide a flexible environment in which complex associations can be represented naturally without an intermediate translation phase from the domain to the storage abstraction. Our model is similar to the OEM used by LORE [2].

The TENTACLE data manipulation subsystem consists of an integrated imperative programming and declarative query language. The language runs directly inside the database system and enables the user to program the database server. This removes the need for gateway programs and wrappers as encountered in more typical database systems. In this respect our system bears some resemblance to database programming languages [4].

We have chosen the world-wide web as an example application to illustrate how the above-mentioned features of the database system can be applied to the task of supporting complex and or poorly structured problem domains.

2 Implementation

The database is implemented as a monolithic system, where the integrated query and host language interpreter is coupled to the storage subsystem so that the entire database runs as a single process, reducing the inter-component communications overhead.

The TENTACLE database system has been written from the ground up. Its lowest layer is a specialised graph storage system which maps the graph components onto disk blocks. The services provided by this layer are used by the graph management system to provide basic graph creation and manipulation services. On top of the graph management layer we have built a re-entrant parser capable of concurrently interpreting several programs written in the combined programming and query language.

3 Application

The world wide web is the largest networked hypertext system. Surprisingly its interaction with typical databases has so far been quite limited—almost all hypertext documents are stored on a conventional file system and not in a database.

In the cases where databases are indeed accessible via the world wide web, the interaction between the web and the database is not direct—in almost all cases such "web-enabled" databases are simply regular databases accessible through a web-based gateway (see CGI [1]). Such an arrangement may not only cause performance problems (a result of the interprocess communications overheads of web server, gateway(s) and database), but also result in impedance mismatches, since usually the database and web use different data models (relational for the database, network for the web server).

Our system attempts to address these issues by moving the web server into the TENTACLE database system. For each incoming connection, TENTACLE activates a program fragment stored in the database itself which provides the web server functionality (by implementing parts of the HTTP [3] protocol). This program uses the incoming request (URL) to traverse the database graph and sends matching nodes or materialised information to the client. Since the mapping from the graph structure to the world wide web is reasonably direct, there is no need to perform cumbersome or costly rewriting operations.

References

1. Common gateway interface (CGI) specifications. http://hoohoo.ncsa.uiuc.edu/cgi/.
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997.
3. T. Berners-Lee, R. Fielding, and F. H. Hypertext transfer protocol HTTP/1.0. http://ds.internic.net/rfc/rfc1945.txt, May 1996.
4. S. M. Clamen. Data persistence in programming languages: A survey. Technical Report 155, School of Computer Science, Carnegie Mellon University, May 1991.

A Toolkit to Facilitate the Querying and Integration of Tabular Data From Semistructured Documents

L.E. Hodge, W.A. Gray and N.J.Fiddian
Department of Computer Science
University of Wales, Cardiff, UK
e-mail: {scmlehlwaglnjf}@cs.cf.ac.uk

Abstract. The advent of the World Wide Web (WWW) has meant there is a growing need to link information held in databases and files with information held in other types of structure [2]. This poster presents a toolkit which enables information held in tables within documents to be linked with data held in conventional databases. The toolkit addresses the problems of:-

- extracting tabular data from various types of document e.g. semi-structured text [4], HTML and spreadsheets
- providing a standard interface to these tables via a standard query language
- extracting useful table meta-data from the table and accompanying text.

Tables are a very effective way of presenting information. Because of their flexibility and ease of reading, tables have long been used as a method of presenting information in a clear and concise manner. With the introduction of the Internet, large bases of unstructured textual information have become available, many of which contain information in a tabular form, an example being results from scientific studies appearing within a paper. There is a growing awareness of this information and the need to link it with more conventionally held information in databases if the full potential of this information is to be realized.

The problem with tabular information is that although the nature of its presentation makes it easy for the reader to understand, it is not possible to access and utilize this information automatically in a computer application and so link it with other information. Part of our research is to produce a set of tools that will facilitate the generation of wrappers [1,3] for diverse tables embedded in various types of document. These wrappers will enable access to the information contained within the tables via a query language such as SQL and so enable them to be linked with other information, particularly that held in relational databases.

When designing and creating a table to represent a set of information, a table designer may need to modify the information in order to present it in the most useful way. In

performing these transformations the designer uses knowledge about the information to alter the way that it is represented. This can be as simple as introducing a 'total' column by adding two columns or converting a percentage mark into a grade. In other cases, complex formulae may be applied to derive the results required for display. Although the designer finally has the table in the form that he wishes it to be displayed it would be useful for us to know how the source information has been manipulated to produce the information displayed in the table. In some cases such information is available in the tables meta-structure (e.g. in a spreadsheet), where as in others it is buried in accompanying text.

As well as developing tools to provide access to information contained within tables, we are developing tools that can extract meta-data from the tables and any surrounding text in a document. This meta-data will essentially consist of any formulae and constraints relating to entries in the table and any information about the representation being used. The problem is that this kind of useful information is usually lost or unavailable. Some types of tables are more useful than others because they store formulae and constraints as part of their definition (e.g. spreadsheets), whereas in other representations of tables such information is simply discarded. Fortunately, information describing how columns or rows of the table are calculated can often be found in the text accompanying the table.

In addition to the tools allowing access to tables embedded within textual documents, we are developing tools to allow access to tables in the form of spreadsheets. Spreadsheets are probably the most common source of tabular data and have a high level of structure. When dealing with spreadsheets, meta-data such as formulae and constraints are easy to locate because this type of information is embedded within the spreadsheet itself. When it comes to textual documents we will need to use techniques based on NLP and data mining to locate and extract the relevant formulae and constraints from the accompanying text – a much more challenging task.

References

[1]Wrapper Generation for Semi-structured Internet Sources. Ashish, N & Knoblock, C. University of Southern California.

[2]Learning to Extract Text-based Information from the World Wide Web. Soderland, S. University of Washington. In Proceedings of Third International Conference on Knowledge Discovery and Data mining (KDD-97).

[3]Wrapper Induction for Information Extraction. Kushmerick, N, Weld, D, Doorenbos, R. Proceedings of IJCAI 97.

[4]Using Natural Language Processing for Identifying and Interpreting Tables in Plain Text. Douglas. S, Hurst. M, Quinn, D. December 1994.

Parallel Sub-collection Join Algorithm for High Performance Object-Oriented Databases

David Taniar° J. Wenny Rahayu+

° Monash University - GSCIT, Churchill, Vic 3842, Australia
+ La Trobe University, Dept. of Computer Sc. & Comp. Eng., Bundoora, Vic 3083, Australia

1 Introduction

In *Object-Oriented Databases* (OODB), although path expression between classes may exist, it is sometimes necessary to perform an explicit join between two or more classes due to the absence of pointer connections or the need for value matching between objects. Furthermore, since objects are not in a normal form, an attribute of a class may have a collection as a domain. Collection attributes are often mistakenly considered merely as set-valued attributes. As the matter of fact, *set* is just one type of collections. There are other types of collection. The Object Database Standard *ODMG* (Cattell, 1994) defines different kinds of collections: particularly *set*, *list/array*, and *bag*. Consequently, object-oriented join queries may also be based on attributes of any collection type. Such join queries are called *collection join queries* (Taniar and Rahayu, 1996).

Our previous work reported in Taniar and Rahayu (1996, 1998a) classify three different types of collection join queries, namely: *collection-equi join*, *collection-intersect join*, and *sub-collection join*. In this paper, we would like to focus on sub-collection join queries. We are particularly interested in formulating a parallel algorithm based on the sort/merge technique for processing such queries. The algorithms are non-trivial to parallel object-oriented database systems, since most conventional join algorithms (e.g. hybrid hash join, sort-merge join) deal with single-valued attributes and hence most of the time they are not capable of handling collection join queries without complicated tricks, such as using a loop-division (repeated division operator).

Sub-collection join queries are queries in which the join predicates involve two collection attributes from two different classes, and the predicates check for whether one attribute is a sub-collection of the other attribute. The sub-collection predicates can be in a form of *subset*, *sublist*, *proper subset*, or *proper sublist*. The difference between proper and non-proper is that the proper predicates require both join operands to be properly sub-collection. That means that if both operands are the same, they do not satisfy the predicate. The difference between subset and sublist is originated from the basic different between sets and lists (Cattell, 1994). In other words, subset predicates are applied to sets/bags, whereas sublists are applied to lists/arrays.

2 Parallel Sort-Merge Join Algorithm for Sub-collection Join

Parallel join algorithms are normally decomposed into two steps: *data partitioning* and *local join*. The partitioning strategy for the parallel sort-merge sub-collection join query algorithm is based on the *Divide and Partial Broadcast* technique.

The Divide and Partial Broadcast algorithm proceeds in two steps. The first step is a *divide* step, where objects from both classes are divided into a number of partitions. Partitioning of the first class (say class *A*) is based on the first element of the collection (if it is a list/array), or the smallest element (if it is a set/bag). Partitioning the second class (say class *B*) is exactly the opposite of the first partitioning, that is the partitioning is now based on the last element (lists/arrays) or the largest element (sets/bags).

The second step is the *broadcast* step. In this step, for each partition *i* (where *i*=1 to *n*) partition *Ai* is broadcasted to partitions *Bi* .. *Bn*. In regard to the load of each partition, the load of the last processor may be the heaviest, as it receives a full copy of *A* and a portion of *B*. The load goes down as class *A* is divided into smaller size (e.g., processor 1). Load balanced can be achieved by applying the same algorithm to each partition but with a reverse role of *A* and *B*; that is, divide *B* based on the first/smallest value and partition *A* based on the last/largest value in the collection.

After data partitioning is completed, each processor has its own data. The join operation can then be carried out independently. The local joining process is made of a simple sort-merge and a nested-loop structure. The sort operator is applied to each collection, and then a nested-loop construct is used in joining the objects through a merge operator. The algorithm uses a nested-loop structure, because of not only its simplicity but also the need for all-round comparisons among all objects.

In the merging process, the original join predicates are transformed into predicate functions designed especially for collection join predicates. Predicate functions are the kernel of the join algorithm. Predicate functions are boolean functions which perform the predicate checking of the two collection attributes of a join query. The join algorithms use the predicate functions to process all collections of the two classes to join through a nested-loop. Since the predicate functions are implemented by a merge operator, it becomes necessary to sort the collections. This is done prior to the nested-loop in order to avoid repeating the sorting operation.

References

Cattell, R.G.G. (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

Taniar, D., and Rahayu, W., "Object-Oriented Collection Join Queries", *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems TOOLS Pacific'96 Conference*, Melbourne, pp. 115-125, 1996.

Taniar, D. and Rahayu, J.W., "A Taxonomy for Object-Oriented Queries", a book chapter in *Current Trends in Database Technology*", Idea Group Publishing, 1998a (in press).

Taniar, D. and Rahayu, J.W., "Parallel Collection-Equi Join Algorithms for Object-Oriented Databases", *Proceedings of International Database Engineering and Applications Symposium IDEAS'98*, IEEE Computer Society Press, Cardiff, UK, July 1998b (to appear).

Component DBMS Architecture for Nomadic Computing

J.A. McCann, J.S. Crane

High Performance Extensible Systems Group
Computer Science, City University, London, UK
{jam, jsc}@cs.city.ac.uk

Abstract. Current database management system technology is unable to provide adequate support for Nomadic Computing. Mobile applications require systems which are sufficiently lightweight and customisable to provide high performance while consuming minimal power, yet extensible enough to adapt to a constantly changing environment. Current DBMS architectures do not provide this level of customization or adaptability. In this paper we introduce Component-Based Database Management Systems (CBDMS) and discuss their suitability for mobile computing.

1. DBMS Architectures are Obsolete?

New technology is placing greater expectations on DBMS. That is, we anticipate greater use of wide-area networks, heterogeneous platforms, information sharing, higher flexibility and dynamic system modification such as 'plug and play' and hot swap. Historically, enhancing features were just added to the DBMS kernel thickening it and reducing performance. This solution will not work for mobile DBMS applications as the mobile unit is very limited in resources .

Industry, has identified a number of key application areas which would benefit from mobile DBMS. These are: healthcare, sales force automation, tourism and transportation. In particular, sales force automation allows the salesperson to use a unit connected through a mobile network to a set of databases. Furthermore, they can quickly configure complex customer solutions and customised financing options *while with the customer.* This type of operation requires that the mobile client can *update* the DBMSs as the transaction occurs which is still *not* being addressed by the DBMS community.

Performance and adaptability are key requirements of mobile data processing environments which existent DBMS architectures fail to provide. These present new challenges in areas such as query processing, data distribution and transaction management which are being re-thought in terms of performance, battery life and less reliable communication systems.

2. A New Adaptive DBMS Architecture Based on Components

To provide the amount of flexibility to enable mobile data processing, we require a combination of a lightweight yet extensible DBMS. We look to the history of operating systems for an answer. Early monolithic and micro-kernel operating systems were limited in extensibility and performance, so research looked at extensible kernels, and more recently component-based operating systems [Kostkova96], as a solution.

Current DBMS technology is monolithic and is neither lightweight nor flexible enough to support mobile computing. There has been attempts to make DBMS architectures more lightweight, or to be extensible, [Boncz94], however they all are unsuitable for general DBMS applications. Therefore, DBMS need to be componentised where only the components required for a particular operation are loaded, saving performance and power costs. Furthermore, a component DBMS can extend its functionality on demand -- it binds its components at run-time, adapting to new environments. For example, the movement from a wireless network to a fixed network can trigger a new resource manager or query optimiser to be dynamically loaded as the system is no longer constrained by battery power and wireless communications. Furthermore, as processing is componentised, its migration between mobile network cells becomes more fluid.

3. Our Work and Conclusion

In this paper we have shown that to balance performance efficiency, power efficiency and cost efficiency in nomadic computing, we need more than new transaction processing and query optimisation strategies. For a DBMS to operate efficiently in a mobile environment it needs to be lightweight and customisable, providing high performance while consuming minimal power. Furthermore, the architecture must adapt to a constantly changing environment. We believe that an alternative DBMS architecture based on components is a viable solution. To this end we are currently implementing a CDBMS, which is already beginning to demonstrate its level of real-time configurability and improvement in performance, for not only mobile applications but more traditional applications.

4. References

Boncz P., Kersten M.L., 'Monet: an impressionist sketch of an advanced database system', BIWIT'95-. Basque Int. Workshop on Information Technology Spain, July 1995.

Kostkova P., Murray K., Wilkinson T.: Component-based Operating System, 2nd Symposium on Operating Systems Design and Implementation (Seattle, Washington, USA), October 1996

ITSE Database Interoperation Toolkit

W Behrendt[1], N J Fiddian[1], W A Gray[1], A P Madurapperuma[2]

[1] Dept. of Computer Science, Cardiff University, UK
{Wernher.Behrendt, N.J.Fiddian, W.A.Gray}@cs.cf.ac.uk

[2] Dept of Computer Science, University of Colombo, Sri Lanka
ajith@cmb.ac.lk

Abstract. The ITSE system (Integrated Translation Support Environment) is a toolkit whose architecture is aimed at enabling the exchange of data between different database interoperation tools as well as the dynamic addition of new tools. Each of the tools addresses a specific set of interoperation problems and achieves interoperation using an intermediate semantic representation to map from source to target expressions.We give a brief overview of the system, the experiences that led to its specific architecture, and some scenarios in which the integrated use of tools would be beneficial. **Keywords**: Tools, Interoperation, Distributed DBMS.

System evolution requirements and tool support. Many organisations require legacy database systems to co-operate alongside modern DBMSs to support business functions. These organisations require system evolution methodologies and tools to evolve systems in their current, heterogeneous setup. The need for appropriate interoperation tools and the lack of integration between such tools is expressed by Brodie and Stonebraker [1]. Typical tasks in system evolution are the analysis of existing database schemas to ensure the conceptual models still reflect business practice; interoperation of new applications with legacy systems; the migration of these older systems to newer platforms, often coupled with a need for extensive re-design; the decoupling of remote user interfaces from monolithic, central information systems, with the added requirement of transparent access to heterogeneous information systems; or the re-interpretation of business data from different angles - one of the main drivers of the development of data warehouses.

The ITSE project ('95-'98) investigates an integration framework for a set of database interoperation tools which have diverse functionality but use a common construction paradigm by viewing interoperation as a translation task between formal languages which stand in a semantic equivalence relation to one another. The ITSE toolkit is a second-generation system providing a framework for these first-generation database interoperation tools built over the last ten years. The new system improves on previous techniques and provides facilities for data interchange between components of the toolkit. This ability is a requirement for cost-effective restructuring of information systems because it adds to the automation of otherwise error-prone design and implementation tasks for which little methodological support has been offered to date.

Architecture. The four main components of the ITSE toolkit are the *Interoperation Workbench* which houses individual interoperation tools (e.g. schema transformers, query translators, analysis and visualisation tools, etc); *Access Tools* utilizing call-level

interfaces to databases (including O/JDBC); the *Configurer* - a systems management tool which declares data sources to the ITSE system, and allows administrators to define which of the workbench tools are available in a specific setup; *Metatools* - a set of utilities to declare and bind new tools to the toolkit. Metatools are aimed at developer organisations whereas Configurer and Workbench are aimed at system administrators or analysts. An extensible client/server architecture allows several modes of operation for the tools: via graphical user interfaces or a scripting language, or via an agent communication language in which interoperation tasks can be requested remotely.

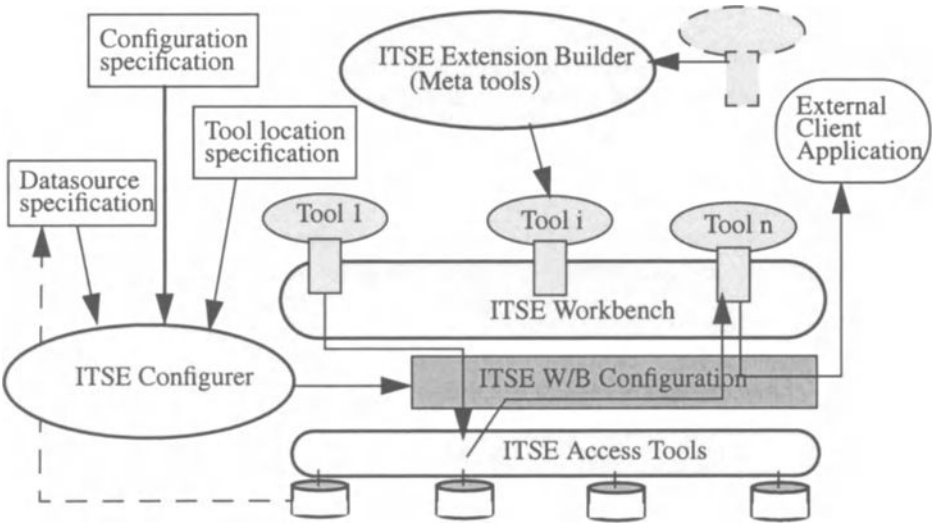


Fig.1. Top-level Functionality of the ITSE Interoperation Toolkit

Prototype implementation. ITSE server modules are implemented in BinProlog [2] with client GUIs written in Java. Connections to network-aware DBMSs are made using either ODBC and JDBC interfaces, or connectivity software developed by ourselves. The workbench includes tools for schema and query translation, as well as visualisation. Incorporating more workbench tools and connecting to different DBMSs is an ongoing process. A set of databases using Oracle, UniSQL, Versant and P/FDM forms a testbed for experimentation and development of further interoperation tools.

Ongoing and further work. The next step is to set up typical usage scenarios to test the usability and utility of the integrated tools. Examples for such scenarios could be schema analysis, on-line query translation, or MDBMS view integration. Complex scenarios include legacy system migration tasks and data warehousing. We also work on embedding Prolog-engines in C++ or Java for CORBA-based distributed services [3].

References

[1] Brodie M L, Stonebraker M, Migrating Legacy Systems, Morgan Kaufmann Publishers, 1995.

[2] ftp://clement.info.umoncton.ca/BinProlog

[3] http://www.omg.org/about/wicorba.htm

Ontological Commitments for Multiple View Cooperation in a Distributed Heterogeneous Environment

A-R. H. Tawil, W. A. Gray, and N. J. Fiddian

Department of Computer Science
University of Wales College of Cardiff, U. K.
{*Abdel-Rahman.Tawil, N.J.Fiddian, W.A.Gray*}*@cs.cf.ac.uk*

This paper deals with combining multidatabase systems with knowledge representation schemes. The current proposal is based on work done in the University of Manchester on a Medical Ontology [Rec94], database interoperation work at Cardiff University in Wales [DFG96], in addition to much related work in the area of semantic interoperability (e.g,[Wie94,ACHK93,KS94,Gru91]).

We are currently investigating a methodology for achieving interoperability among heterogeneous and autonomous data sources in a framework of knowledge sharing and re-use, based on a multi-view architecture. In this paper we describe a novel approach for integrating a set of logically heterogeneous object oriented (OO) and/or relational databases. It is based on using a view integration language capable of reconciling conflicting local classes and constructing homogenised, customised and semantically rich views. The integration language uses a set of fundamental operators that integrate local classes according to their semantic relationships and it supports *ontological commitments*, e.g., agreements to use a shared domain ontology [1] in a coherent and consistent manner. We view ontological commitments as a very important requirement in our work. In our approach, each view should commit to a particular ontology to enable other views to infer the kinds of concepts they can interpret and deal with. Here, ontological commitment specifies a commitment to the use of a term and to the interpretation of that term as described in a shared domain ontology.

The approach to our multiple-view knowledge sharing and re-use is briefly described in the following way. During the schema integration process, and specifically during the analysis phase of the integration process, the knowledge accrued from identifying the semantic relationships across two schema objects is encapsulated within the generated schema (view). This knowledge is costly to collect and maintain, and is hard to share and re-use. We therefore propose the organisation of this knowledge by levels of semantic granularity (schema, tables and attribute levels), gathering semantic information at progressively more refined levels of schematic heterogeneity. Hence, during the application of each integration operator of our integration language, the integrator is prompted to

[1] A shared ontology is a library of reusable context independent models of the domain

define/verify the explicit assumptions made in the identification of relationships among the integrated objects. These captured assumptions are represented as metadata extracted from the domain specific ontology and associated with each class or attribute of the integrated schema. Also, with each operator applied, a new expression adding this knowledge is compositionally generated using the power of a compositional description logic language.

By abstracting the representational details of the views and capturing their information content, we provide the basis for extending querying facilities to enable the spanning of multiple views based upon the interpretation of the semantic knowledge which *previously* was encapsulated in each view.

The poster will show the architecture of our integration system and the structure and types of tool it supports. We will illustrate how by using domain ontologies we can enrich the database metadata with descriptive domain knowledge. We will also describe how the integration system makes use of this domain knowledge to ease the integration process and to create integrated views which maintain a *declarative, object oriented, structured* models of their domain of expertise and of the domain of expertise of each of their information sources. Thus, providing a level of understanding of the information being integrated.

References

[ACHK93] Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993.

[DFG96] R. M. Duawiri, N. J. Fiddian, and W. A. Gray. Schema integration metaknowledge classification and reuse. In *In proceedings of the 14th British National Conference on Databases (BNCOD14), Edinburgh*, pages 1–17, UK, July 1996.

[Gru91] T. R. Gruber. The role of common ontologies in achieving sharable, reusable knowledge bases. In E. Sandewall J. A. Allen, R. Fikes, editor, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference, Cambridge, MA*, pages 601–602. Morgan Kaufmann, 1991.

[KS94] V. Kashyap and A. Sheth. Semantic based information brokering. In *Proceedings of the 3rd Intl. Conf. on Information and Knowledge Systems*, November 94.

[Rec94] A. Rector. Compositional models of medical concepts: Towards reusable application-independent medical terminologies. In P. Barahona and J. Christensen, editors, *Knowledge and Decisions in Health Telematics, IOS Press*, pages 133–142, 1994.

[Wie94] Gio Wiederhold. Interoperation mediation and ontologies. *Proceeding in international symposium on fifth generation computer systems (FGCS94), Workshop on heterogeneous cooperative knowledge-bases, ICOT, Tokyo, Japan*, W3:33–48, December 1994.

A Repository to Support Transparency in Database Design

Gerrit Griebel[1]**, Brian Lings[2], and Björn Lundell[1]

[1] University of Skövde, Sweden, Department of Computer Science
(bjorn.lundell@ida.his.se)
[2] University of Exeter, UK, Department of Computer Science
(brian@dcs.exeter.ac.uk)

Abstract. In this paper we address aspects of traceability in Information Systems design, particularly with those methodologies utilizing rich, and possibly multiple (behavioural) models. We consider this issue within the broader context of IS development, and focus on the potential for communication among human actors. Specifically, we consider the question of repository design for supporting design activities. We illustrate this through a Web-based visualization tool for Extended Entity-Relationship (EER) modeling represented in a repository based on the Case Data Interchange Format (CDIF). Such a tool offers a degree of independence with respect to CASE tools used within an organization, and validation methods chosen by an organization.

1 Introduction

1.1 Model Transparency

Traceability in Information Systems design, particularly with those methodologies utilizing rich, and possibly multiple (behavioural) models, is a complex issue. The aspect of traceability of concern in this paper, sometimes referred to as modeling transparency, is considered as an absolute requirement for advanced systems development [3].

Within the broader context of IS development it requires correlation of design decisions across different sub-models, and between data sets and meta data, on an ongoing basis. A number of different tools may be utilized, including CASE tools offering ER to SQL mapping and external validation tools. One way forward is the utilization of repository technology for supporting all design and maintenance activities. Such a repository would be designed to store all available modeling information, and all information necessary to allow transparency in the refinement process and with respect to dependencies between models. We have chosen to explore this idea through an investigation of the use of the Case Data Interchange Format (CDIF), initially for defining Extended Entity Relationship (EER) models, and latterly with respect to multiple models in the context of the

** current address: Soester Straße 48, 20099 Hamburg, Germany (gg@tron.ppp.de)

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 19–31, 1998.

F3 methodology [6]. All the work so far has concentrated on the design phase of the life cycle; the support of maintenance activities forms a part of ongoing work. For this study, CDIF was chosen for its extensibility. We note, however, that the investigators within the F3 project had suggested the possible use of CDIF for our purposes: "We strongly suggest the CDIF standard is considered for F3 and its inter-tool communication." [5, p. 39]

We observe two main approaches to enhancing ER with respect to behavioural aspects at the conceptual level. Firstly, some contributions have added constructs to give a particular variant of EER, e.g. rules in [19] (anticipating active databases). A consequence of this approach is an increased complexity within diagrams, something which potentially is a hindering factor for its use in practice. Secondly, others suggest a set of different models to be used in conjunction at the conceptual level (as in F3). A consequence of this approach is that the number of inter-relationships that need to be maintained between related concepts in different models increases, potentially exponentially, with the number of models used.

In addition to 'extended model' and 'multi-model' approaches, we envisage a need for enhanced traceability between the conceptual level models and those models used at earlier stages of the IS development process. Our assumption here is that as the complexity of models increases, so will the variation in expectations and needs between users [13]. To address this difficulty, we anticipate that a meta-model approach (repository) will allow us to handle this complexity, something which we are currently exploring in the context of two (of the total of five) different F3 models with preserved traceability between the Enterprise Model level and the conceptual model.

1.2 Populated Models

We believe that, as a complement to communication at the level of models, techniques for browsing, visualization, animation etc. (see e.g. [4] and [12]) using populated models might significantly improve end-users' understanding of behavioural models.

Sample data or prerecorded data is often available before a development starts; investigation of the domain is facilitated by this data, which is often not used until installation of the developed database software.

When data is incorporated, a modeling tool can provide visual feedback of the dynamic implications of a schema and its integrity constraints (IC). This is useful for understanding an application database, since most DBMS allow predefined behaviour to maintain the integrity of data. This is particularly of value in the context of EER-Modeling, because relational IC can be automatically derived from EER model properties to lose as little model semantics as possible during mapping.

1.3 Aim of the Research

The concern of this work is therefore threefold:

1. To investigate mapping procedures with respect to preservation of EER model semantics
2. to design a repository suitable for storing all model representations and their interrelationships and
3. to propose a visualization architecture based on the repository and target application database to allow for visualization of the meta-data and the data at different levels.

The result is a proposal for a repository based visualization architecture which explicitly relates the application database data to the abstract relational level and to the EER level. Different aspects of the design and the refinements chosen can be traced in a comprehensible fashion. The focus of this work is design. The modeling tool S-Designor [15] was used to build the CDIF repository structure and to generate sample models. Java programs were written to exemplify access and visualization of the repository content [11].

2 The Layered Approach

We consider a three layered approach to designing schemas for DBMS as represented by the three boxes in the middle column of Figure 1.

A central point of this work is to allow not only a transparency of the mapping from EER to relational, but also its inverse. Commonly only the forward mapping from the EER to the relational model is considered in the design process. This is insufficient to support general visual browsing with genuine traceability. Appropriate structures must be found for storing this bidirectional interrelationship (Meta-Schema, left column of Figure 1). Structural mapping procedures are well covered by the literature (as in [9]). With the use of modern tools, most of the mapping can be derived automatically, while the designer can give explicit guidance, for example, on how to migrate candidate keys to referencing relations or how to map supertype/subtype or 1:1 relationships.

We focus on the production of a relational model which reflects the inherent constraints in an EER model with little loss of semantics. Some EER IC properties (e.g. disjoint relationships) cannot be expressed in the abstract relational layer, but can be modeled by trigger sets of a target database system. Therefore access paths from the EER information to the SQL-layer are necessary. Update propagation properties (cascade, restrict, update, set null and set default, comparable with those in [14]) which define the behaviour of the database in case of violation of IC are not included in common EER notations. These relational properties can be supplied at the abstract relational layer and stored in appropriate structures belonging to this layer. Also parameters for mapping the abstract relational model to a target SQL dialect must be supplied and stored.

Current modeling tools often intermix model properties with mapping information or do not include mapping information at all. If the mapping algorithm of a modeling tool does not allow the user to generate a relational model with

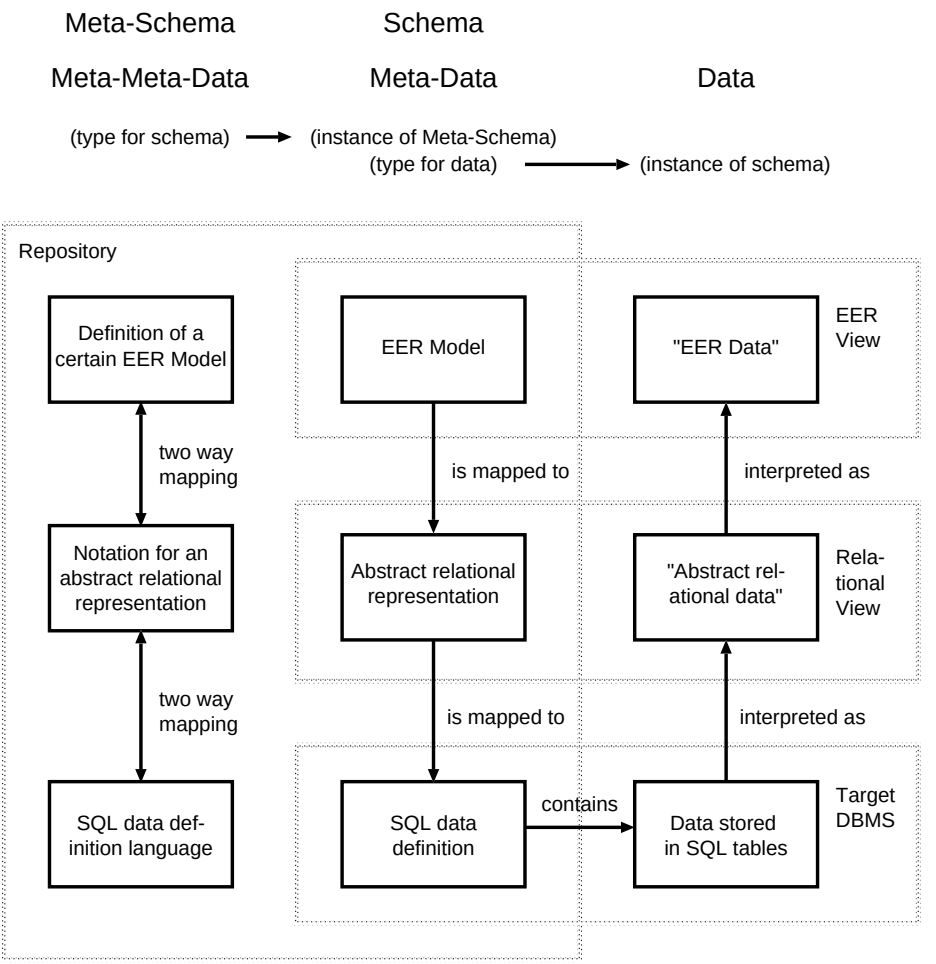


Fig. 1. Meta-Schema, Schema and Data

the desired properties, manual changes might be necessary at the relational level. This will result in a situation where the three layered approach will at the most be used for the initial design, but not during system development.

3 Repository Design

In our opinion, a well designed repository structure is needed to support the storage of all design information including the models and interrelationships between the models. This would allow tight synchronization, and facilitate browsing and the investigation of design implications.

A good test of such a repository would therefore be its use as a basis for visualization of interrelationships within and between meta data and data for an information system. In a broader context such a repository could also act as a unified store for a number of different tools, including schema design tools. Figure 1 depicts the repository content: its structures (left column) and the *type* data it contains (middle column). *Instance* data is not contained in the repository; this resides in the target DBMS (bottom row).

> "Repositories must maintain an evolving set of representations of the same or similar information. [...] Relationships among these representations must be maintained by the repository, so changes to one representation of an object can be propagated to related representations of the same object.", [18, Section 5.3.3],

Interrelationship information in the repository allows the interpretation of data in the target application database at three levels (as depicted in Figure 1): that of the "Target DBMS", that of the "Relational View" and that of the "EER View".

We were looking for standardized repository schemas and have chosen CDIF. As the name suggests it is "a single architecture for exchanging information between CASE tools, and between repositories" which is "vendor-independent" and "method-independent" [8]. It should overcome the problem that no commonly agreed standard repository exists by defining a way of transferring data on the same *subject area* from one tool or repository to another. Case data transfer is not the topic of this project. However, the structures offered by the subject areas *Data Modeling* (DMOD) and *Presentation, Location & Connectivity* (PLAC) were chosen as a basis for a repository structure to store EER and relational models and their graphical representation. The separation between models and their graphical representation supports a modular design. All subject areas are hierarchically structured and make extensive use of inheritance to minimize redundancy. This suggests a direct mapping to a class hierarchy of an object oriented host language. The full range of objects and relationships of DMOD is displayed in Figure 2 and our simplified implemented model is depicted in Figure 3. The associated PLAC design is not depicted.

The DMOD subject area is capable of storing EER- as well as relational models in a single structure. Most objects are used for both layers, some (e.g.

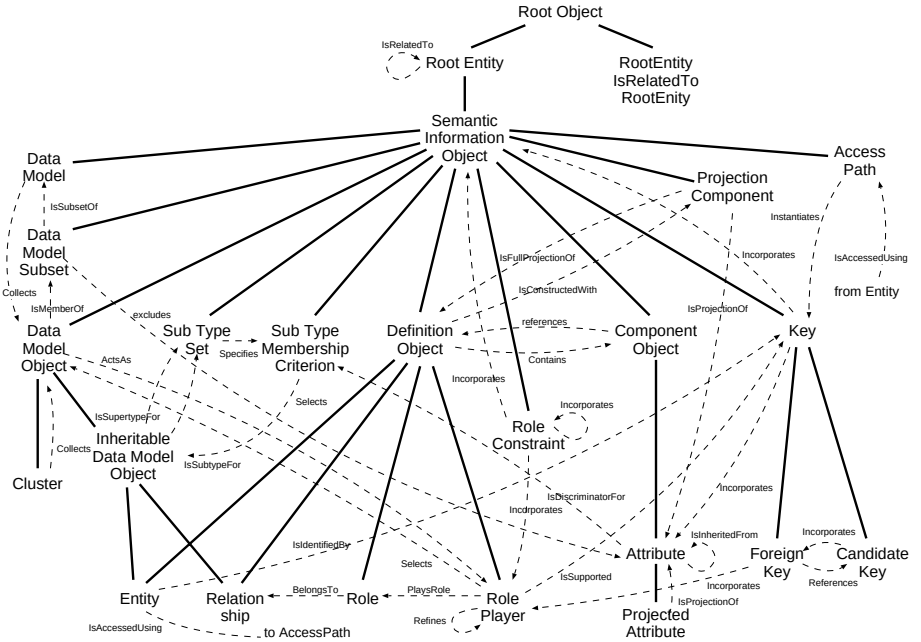


Fig. 2. CDIF-DMOD: overview of objects and relationships

foreign keys or supertype/subtype) only for one. CDIF does not provide structures to relate EER objects to relational objects. We circumvent this problem by extending the subject-area to cope with these correspondences as depicted in Figures 3 and 4. Migration rules for implementing primary and foreign relational keys from EER candidate keys are not included in the model, since we rely on the mapping procedure of the S-Designor modeling tool which does not support them.

In the following we introduce a visualization tool based on the repository, which depicts the different layers shown in Figure 1.

4 Visualization Tool

4.1 Visualization Architecture

Initially it was planned to have a single view for the complete visualization, including data and schema. This would mean, for example, that at the EER level the visualization would include entity instances as well as relationship instances. It would be possible to include this information into a schema only for very limited example models. Scalability is often a problem with such visualization. It is not only a technical or geometric issue of placing graphical elements on a limited two dimensional space but also of comprehensibility for users. This problem led to the overall visualization architecture as laid out in this chapter.

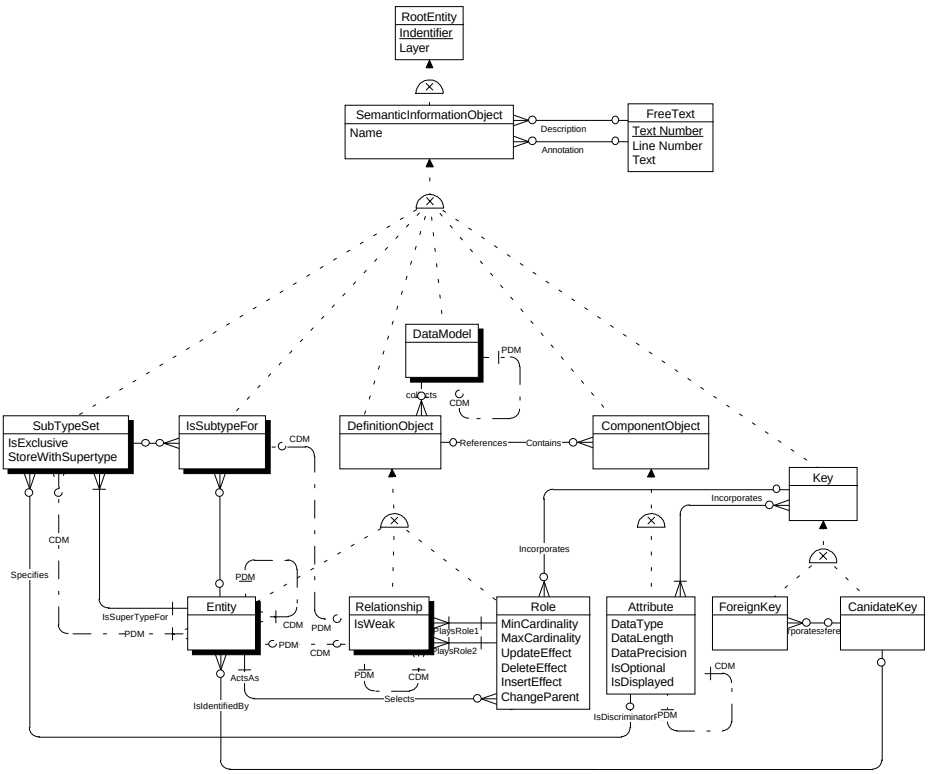


Fig. 3. Conceptual schema of the repository structure

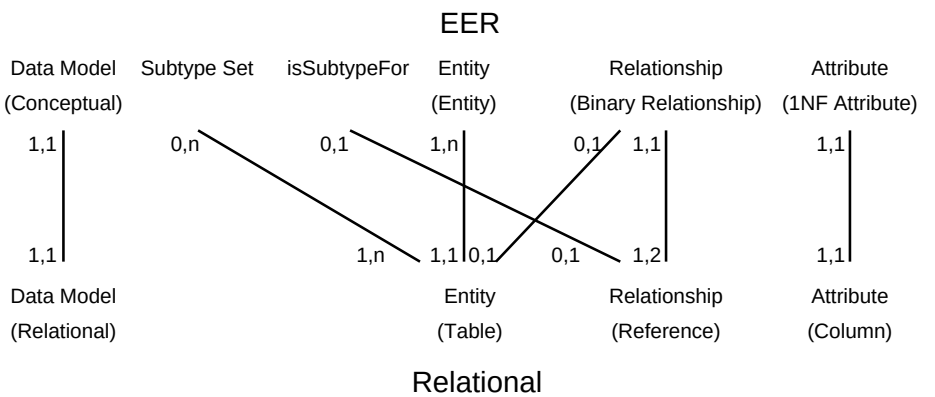


Fig. 4. Mapping-meta-relationships (layer dependent terms in parentheses; numerical cardinalities given (min,max))

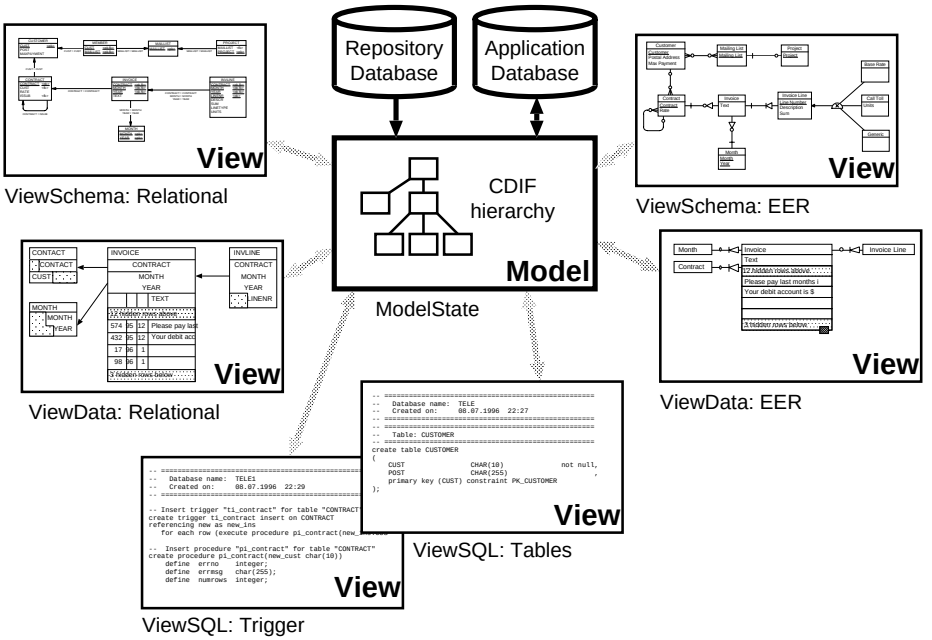


Fig. 5. Model and Views

The basic idea is to split the wealth of displayable information into manageable and comprehensible pieces and at the same time to relate these partial views through visual feedback. Views are deliberately limited to only certain aspects of the whole, as represented by the boxes in Figure 1 (Meta-Schema, Schema, Data combined with EER, relational, SQL). Examples of views appear below.

We make use of the ideas and terminology of the Model-View-Controller (MVC) paradigm which stems from the Smalltalk environment: "The *model*[1] is used to represent the data or knowledge about the application constructed" [2] and the *view* is the output interface, while the *controller* takes inputs from the user and passes them to the *model*. "A major idea in the MVC is the isolation of the model from the view and controller. Because the model normally represents a set of data or knowledge, it has no need to know how the controller or the view operates" [2].

Logically, the model is a server which accepts connections from views and controllers (clients) and communicates with them via a protocol (arrows in Figure 5). The model has direct access to the repository (data model content (DMOD) and its graphical presentation (PLAC)) and the target database, while the views access the repository data only via the connection to the model.

If the user interacts with one view, the result should not only be apparent in this view, but in all open views at the same time. This concerns highlighting

[1] Not to be confused with "data model".

of objects and update visualization. A simple example is a click on an entity in an EER-view and a visual highlighting feedback in all views which display information on the same object. An abstract relational view would highlight the appropriate relation and a SQL-view would highlight the relevant SQL data definition statements. The granularity of highlighting and displaying should be as small as possible, i.e. if a simple attribute should be shown, then only that attribute should be highlighted and not the whole entity or relation.

It should be possible to open new views with a focus on a particular object by clicking on that object in an already existing view. In this way one can browse a model according to the matrix of Figure 1. Relating objects in different views is also of value for managing sub-models with common objects.

4.2 Supported Views

The following short descriptions give an overview of the possible types of view. Firstly, views are introduced which are limited to visualizing the schema (middle column in Figure 1):

Schema: provides a conventional graphical representation of the EER model or relational model without including data. The data for screen layout and textual annotation is taken from the repository (PLAC).

SQL: Since SQL code is generated automatically using templates of the schema design tool it is possible to include markers which allow this view to highlight data definition statements. In this way the SQL code can be browsed by clicking on objects in other views.

Text: "Validation can be done by paraphrasing a conceptual schema in natural language and giving that paraphrase to a user for examination" [4]. The explanations may be automatically derived from the schema semantics or could include textual descriptions. This view reacts to highlighting messages from other windows.

The following views allow visualization and manipulation of instance level data (right column in Figure 1). The schema views above should nevertheless react to messages concerning data manipulation, by using them to highlight relevant schema objects. Different colours can indicate the action performed (for example red=delete, green=insert, blue=update). If a trigger is initiated, its behaviour could be simulated in a single step mode. The **SQL**-view might display a related trigger definition.

Data: This view is most meaningful for visualizing data and update propagation and can be used for EER and relational models. The screen layout is exemplified in Figure 6. Only objects of interest are shown, but browsing allows further schema exploration (upper mouse click in Figure 6). The screen layout can be derived from the schema semantics of the relational layer, since references are directed. Referenced relations appear to the right of referencing relations. The layout of the EER model can be determined with

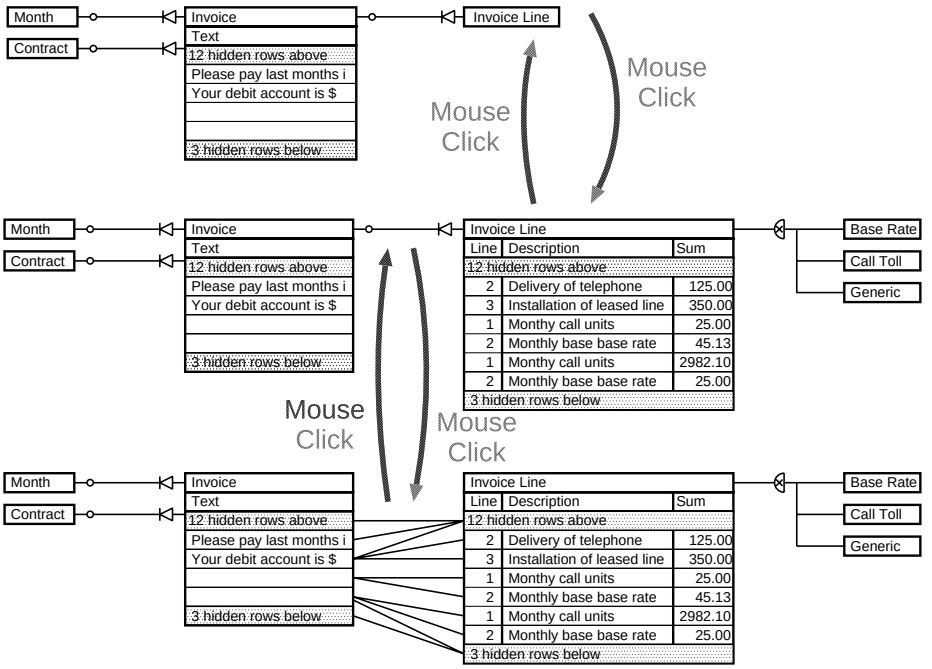


Fig. 6. Data-View example

the aid of the relational representation. Objects may be popped up to show instance level data or popped down to hide it (lower mouse click in Figure 6). Relationship instances as well as data tuples may be graphically inserted, deleted and updated. Submission of data may trigger update propagation, which can be visualized in a single step mode.

Cascade: This passive view is an implementation of the rule trace trees used in [10] and [7]. The first event of a cascade is drawn at the left border as depicted in Figure 7, and all subsequently invoked rules and triggered events are drawn to the right of it, i.e. the flow of time is visualized. Rules may be visually linked to SQL triggers, and events to sets of data tuples.

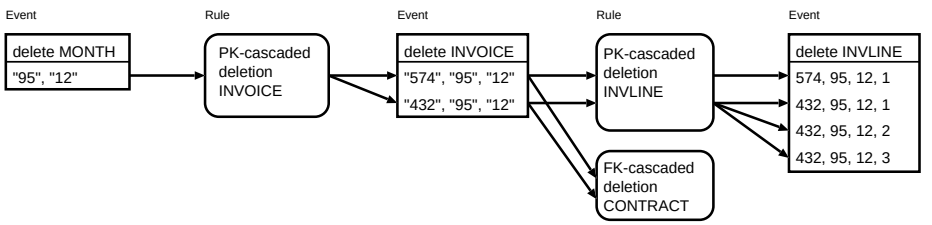


Fig. 7. Sample triggering cascade

Trace: This passive view records the active updates of the database in a textual way comparable with the log file of a DBMS.

Tree: A collapsible tree, as known from file-browsers, is used to access the repository contents in a tabular fashion. It is structured by object type and allows inclusion of instance level data.

4.3 Implementation

The S-Designor database design tool was used to edit an EER-Model and to map it to its relational representation. The tool saves both models, represented as relational tables, in separate plain ASCII-files. The tool was also used to generate tables and triggers for the target DBMS. All files are parsed and loaded into the developed repository. Additionally, the table and trigger SQL-scripts are executed on the target application database.

The visualization tool was implemented using the Java programming language. The tool can be executed within WWW pages[2] assuming that a Java capable browser is used. It becomes a part of the overall presentation. Implemented features are presented next to designed or planned features. The class structure for repository access was completed, but only a selection of views were implemented.

A loader class reads the model via JDBC[3] and instantiates each object as a Java object. Relationships are modeled as object references. Ideally one would use a persistent object oriented programming language based on an object oriented DBMS where "a repository manager would map its object base into labeled directed graphs, where objects are mapped to nodes and relationships are mapped to edges." [1, pp. 710–711]

The model in the MVC sense is a Java class which inherits from the standard class "Observerable" and each view consists of a Java class which inherits from "Observer" to allow the *observerable* object to notify its *observers*. The protocol for communication between views and the model consists of public methods in the model class.

Application data is accessed from the relevant meta-data classes. Update propagation must be simulated by the classes, since we want to execute and visualize update propagation in a step by step fashion. Java provides a rollback mechanism through exceptions to model transactions. We use update propagation properties which are included in the repository and do not consider, for example, SQL-triggers. Visualization could be extended to arbitrary active behaviour if a direct interface to the rule manager of a DBMS existed. This would also allow visualization of updates executed by application programs.

[2] URL http://www.his.se/ida/research/groups/dbtg/demos/griebel/
[3] Java Database Connectivity

5 Conclusions

In this paper we have described work in the area of repository design for Information Systems development. This work targets tool-independent storage of design information, facilitating transparency within and between stages in the refinement process. The focus of the paper is on a specific facet, namely a repository structure for the multi-level visualization of dependencies in populated models using EER to Relational refinement methods.

Many variations of ER/EER models have been proposed in the literature, and it is not uncommon for a commercial modeling tool to provide support for more than one such variation. For example, both LogicWork's tool ERwin/ERX and Powersoft's tool PowerDesignor support more than one EER dialect [16] [17].

Using a meta-model approach (i.e. a generic ER/EER meta-schema) as exemplified by the system described here, a semantically expressive meta-model could, at least in principle, support any variation of the ER/EER Model.

Populated models are used in many database oriented application development environments. For example, Borland's Delphi environment enables instant access to data kept in an underlying database during the design process. We consider them fundamental in improving feedback in the design phase for behavioural models.

The unification of design data within a repository facilitates the development of generic tools, with presentation levels tailored to organizational standards. The tool presented here is a visualization tool, allowing multiple, interlinked views of populated models to be simultaneously displayed. The tool is itself of interest in exploring useful facilities for visual browsing of designs. However, it primarily fulfils the role of a practical test of the repository design.

References

1. Philip A. Bernstein and Umeshwar Dayal. An Overview of Repository Technology. In *Proceedings of the 20th VLDB Conference*, pages 705–713. VLDB Conference, 1994. 29
2. John R. Bourne. *Object-Oriented Engineering — Building Engineering Systems Using Smalltalk-80*. Richard D. Irwin, Inc., and Aksen Associates, Inc., 1992. 26, 26
3. S. Brinkkemper. Integrating diagrams in CASE tools through modelling transparency. *Information and Software Technology*, 35(2):101–105, February 1993. 19
4. Janis A. Bubenko and Benkt Wangler. Research directions in conceptual specification development. Technical Report 91-024-DSV, SYSLAB, Department of Computer Science and Systems Science Stockholm University and SISU — Swedish Institute for Systems Development, November 1991. 20, 27
5. J. A. Bubenko jr, R. Dahl, M. R. Gustafsson, C. Nellborn, and W. Song. Computer Support for Enterprise Modelling and Requirements Acquisition. Technical Report Deliverable 3-1-3-R1 Part B, Swedish Institute for Systems Development (SISU), November 1992. 20

6. Janis A. Bubenko jr. Extending the Scope of Information Modeling. In A. Olivé, editor, *Fourth International Workshop on the Deductive Approach to Information Systems and Databases*, pages 73–97. Lloret de Mar, Costa Brava, September 20–22 1993. 20
7. S. Chakravarthy, Z. Tamizuddin, and J. Zhou. A visualization and explanation tool for debugging eca rules in active databases. Technical Report UF-CIS-TR-95-028, University of Florida, November 1995. 28
8. Johannes Ernst. Introd. to CDIF, 1997. URL http://www.cdif.org/intro.html. 23
9. Christian Fahrner and Gottfried Vossen. A Survey of Database Design Transformations Based on the Entity-Relationship Model. *Data & Knowledge Engineering*, 15(3):213–250, 1995. 21
10. Thomas Fors. Visualization of Rule Behavior in Active Databases. In S. Spaccapietra and R. Jain, editors, *Visual information management: Proceedings of the third IFIP 2.6 working conference on visual database systems*, pages 215–231. Chapman & Hall, 1995. 28
11. Gerrit Griebel. Repository Support for Visualization in Relational Databases. Master's thesis, University of Skövde, September 1996. 21
12. Björn Lundell. Penning a Methodology: A Classroom example used as a framework for reasoning about Information Systems Development. In N. Jayaratna, T. Wood-Harper, and B. Fitzgerald, editors, *Proceedings of the 5th BCS Conference on "Training and Education of Methodology Practitioners and Researchers"*. The British Computer Society: Specialist Group on Information Systems Methodologies, Preston, UK, August 1997. 20
13. Björn Lundell and Brian Lings. Expressiveness within Enhanced Models: An Infological Perspective. In S. W. Liddle, editor, *Proceedings of the ER'97 Workshop on Behavioral Models and Design Transformations: Issues and Opportunities in Conceptual Modeling*. UCLA, Los Angeles, California, November 1997. URL http://osm7.cs.byu.edu/ER97/workshop4/. 20
14. Jim Melton and Alan R. Simon. *Understanding the New SQL: A Complete Guide*. Morgan Kaufmann, 1993. 21
15. Powersoft. *S-Designor DataArchitect 5.0 User's Guide*. Powersoft Corp., 1996. 21
16. Robin Schumacher. ERwin/ERX 3.0. *DBMS*, 10(11):31–32, October 1997. URL http://www.dbmsmag.com/9710d08.html. 30
17. Robin Schumacher. PowerDesigner 6.0. *DBMS*, 10(11):34,36,49–50, October 1997. URL http://www.dbmsmag.com/9710d09.html. 30
18. Abraham Silberschatz, Michael Stonebraker, and Jeffrey D. Ullman. Database Research: Achievements and Opportunities Into the 21st Century. *SIGMOD Record*, 25(1):52–63, 1996. 23
19. Asterio K. Tanaka. *On Conceptual design of Active Databases*. PhD thesis, Georgia Institute of Technology, December 1992. 20

Constraint and Data Fusion in a Distributed Information System *

Kit-ying Hui ** and Peter M. D. Gray

Department of Computing Science, King's College
University of Aberdeen, Aberdeen, Scotland, UK, AB24 3UE
{khui|pgray}@csd.abdn.ac.uk

1 Abstract

Constraints are commonly used to maintain data integrity and consistency in databases. This ability to store constraint knowledge, however, can also be viewed as an attachment of instructions on how a data object should be used. In other words, data objects are annotated with declarative knowledge which can be transformed and processed.

This abstract describes our work in Aberdeen on the fusion of knowledge in the form of integrity constraints in a distributed environment. We are particularly interested in the use of constraint logic programming techniques with off-the-shelf constraint solvers and distributed database queries. The objective is to construct an information system that solves application problems by combining declarative knowledge attached to data objects in a distributed environment. Unlike a conventional distributed database system where only database queries and data objects are transported, we also ship the constraints which are attached to the data. This is analogous to the use of footnotes and remarks in a product catalogue describing the restrictions on the use of specific components.

2 Constraints, Databases and Knowledge Fusion

We employ a database perspective where integrity constraints on a solution database are used to describe a design problem. This solution database can be visualised as a database storing all the results that satisfy the design problem. The process of solving the application problem, therefore, is to retrieve data objects from other databases and populate this solution database while satisfying all the integrity constraints attached to the solution database and all data objects. In practice, this solution database may not hold any actual data but provides a framework for specifying the problem solving knowledge.

To facilitate knowledge reuse, we utilise a multi-agent architecture (figure 1) with KQML-speaking software components. The user inputs the problem specifications through a user-agent in the form of constraints. These piece of knowledge, together with other constraint fragments from various resources, are sent

* This research is part of the KRAFT research project [1] funded by EPSRC and BT.
** Kit-ying Hui is supported by a grant from BT.

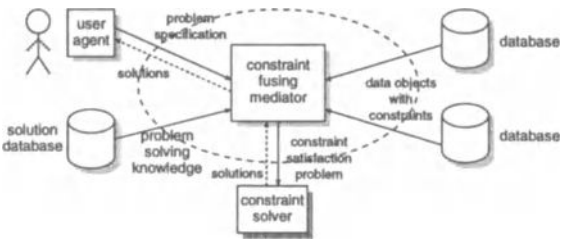


Fig. 1. *Constraint fragments are fused by the mediator to compose a constraint satisfaction problem which is solved by a constraint solver.*

to a constraint fusing mediator which then composes the overall description as a constraint satisfaction problem.

3 Distributed Database Queries and Constraint Solving

The knowledge fusion process produces a declarative description of the constraint satisfaction problem (CSP) which is independent of the problem solving strategy. Given a constraint solver and a set of database servers, the choice of using constraint logic programming (CLP) techniques or generating distributed database queries depends on the complexity of the CSP, capabilities of the constraint solver and efficiency of the database servers. A constraint solver is more efficient in solving complex constraints while database queries can reduce the initial variable domains and relieve network traffic by removing incompatible solutions. A good solving strategy, therefore, should fully utilise the advantages of the two paradigms while complementing their weaknesses by each other.

4 Current Status and Future Plan

We have implemented an early prototype of the system with agents programmed either in Java or Prolog, and communicating through a subset of KQML. Constraints are expressed in the CoLan language [2] for P/FDM. Constraint transformation and fusion are based on first order logic and implemented in Prolog. At the moment we are experimenting the use of a constraint logic programming system, ECLiPSe, to solve the composed CSP. Work is still in progress to divide labour between distributed database queries and constraint solving.

References

1. P. Gray et al. *KRAFT: Knowledge fusion from distributed databases and knowledge bases.* In R. Wagner, editor, Eighth International Workshop on Database and Expert System Applications (DEXA-97), pages 682-691. IEEE Press, 1997.
2. N. Bassiliades and P. M. D. Gray. *CoLan: A function constraint language and its implementation.* Data & Knowledge Engineering 14 (1994) pp203-49. Also `http://www.csd.abdn.ac.uk/~pfdm`.

Observation Consistent Integration of Views of Object Life-Cycles *

Günter Preuner and Michael Schrefl

Department of Business Information Systems
University of Linz, A-4040 Linz, Austria
{preuner, schrefl}@dke.uni-linz.ac.at

Abstract. A commonly followed approach in database design is to collect user views on the database and to develop the conceptual schema of the database by integrating these views.
The design of object-oriented databases involves the design of object behavior next to the design of object structure. Object-oriented design notations usually represent object behavior at two levels of detail: by activities (which correspond to methods at the implementation level) and by object life-cycles which model the dynamics of objects over their lifetime.
This paper discusses the integration of views of object life-cycles that are represented by behavior diagrams, which model the behavior of objects by activities and states corresponding to transitions and places of Petri nets. The presented approach is particularly relevant for the design of business processes, a major application domain of object-oriented database systems.

1 Introduction

The design of an object-oriented database schema involves the definition of the structure and the behavior of objects. Behavior is specified by methods and object life-cycles, which indicate processing states of objects and specify in what order methods may be invoked. Often the database schema is not modeled by a single person, but several user-specific view schemas are defined independently and are integrated later into a single schema.

A major application domain for object-oriented database systems is the business area. There, objects represent business cases and object life-cycles represent business processes. Such as the conceptual schema of a database is often developed from different user views on the database, a business process may be constructed from the views on the business process in different suborganizations.

Consider, for example, a hotel, where the business objects are room reservations: The reception deals with activities check-in and charging services; the booking department considers booking a room and cashing a deposit, etc. Activity check-in will be considered in both views as the guest checks in at the

* This work was partly supported by the EU under ESPRIT-IV WG 22704 ASPIRE (Advanced modeling and SPecification of distributed InfoRmation systEms).

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 32–48, 1998.

reception and the booking department checks whether the guest arrives at the agreed day.

As each view represents only those aspects of the business process that are relevant for a certain user-group, each view may cover only a subset of the aspects of the business process and each view may represent aspects of the business process at a different level of granularity than another view.

The aim of the integration process is to develop an integrated business process from the given views such that if business objects are processed according to the description of the integrated business process, their processing may be *observed* as a correct processing according to each view of the business process *(observation consistent integration).*

In [3], we introduced a two-schema architecture for modeling *business processes* and *workflows,* which represent *external* and *internal* business rules, respectively. External business rules restrict the processing of business objects due to natural facts or law (e.g., check-in is always performed before check-out, but payment and use of a room may occur in any order). Internal business rules exist only due to intra-organizational rules and represent the internal "flow of work" (e.g., payment has to be performed *before* check-in if the guest is unknown).

The distinction between business processes and workflows provides for *workflow transparency,* which allows to change the flow of work without compromising the correctness of the business process. In this paper, we treat the integration of views of repetitive and predictable business processes represented by object life-cycles; we do not consider the integration of workflows and omit definition of exception handling and ad-hoc processing of business objects.

View integration differs from integration of heterogeneous autonomous data sources, which is mainly treated in the realm of federated database systems [13]. In the former, there is only *one* object for each real-world business case; this object behaves according to the *integrated business process.* The views do not handle objects of their own, but only *observe* the processing of objects. In the latter, the global schema is an integration of several local schemas where each object of the real world may be represented by a set of local objects, one in each local autonomous database, and by a global object whose properties are derived from the local ones. Despite their difference, there are many similarities between the integration of views and the integration of independent heterogeneous database schemas, e.g., the detection of heterogeneities between the given schemas.

A number of approaches to schema integration consider only the structure of objects (see e.g. [4] for a survey). Some approaches consider methods, too [14,16], but little work has been done so far concerning the integration of *object life-cycles.* In [5], behavior is modeled in the form of event structures and *conflict-freeness* is defined as a necessary correctness criterion to integrate schemas. Conflictfreeness means that the rules of one schema together with a set of integration assertions do not restrict the models of the other schema. Another approach [6] discusses view integration based on statecharts where views may be connected in

some way (e.g., sequentially, parallely, alternatively) to an integrated schema. In a previous paper [9], we discussed the integration of business processes that treat *disjoint sets* of business objects in autonomous databases, where the common behavior can be observed at the global level.

The work presented in this paper goes beyond the approaches of [5] and [6] as we introduce a detailed integration process that may be applied to views and that produces step by step an integrated business process. The integration of views of business processes is different from the integration of local business processes as described in [9] for two reasons: (1) The former concerns view integration, the latter the integration of similar, autonomously executed business processes. (2) The extensions of the given schemas are the same in the former, but disjoint in the latter.

We consider the integration of views of business processes that are modeled using *Object/Behavior Diagrams* (OBD) [7]: The static properties of objects are defined in Object Diagrams, whereas their life cycle is modeled in Behavior Diagrams, which are based on Petri nets [8].

We will define the correctness of the integrated schema with respect to the given view schemas by using results on the *observation consistent specialization of object life-cycles,* which has been treated in several approaches based on Petri nets [10,11,15]. Our work goes beyond these approaches as not a *single* object life-cycle is specialized but a common subtype of a *set of object life-cycles* (i.e., the views) has to be defined where correspondences and heterogeneities between the views have to be determined. Specialization of object life-cycles comprises *refinement*, where activities and states are refined to a finer level of granularity, and *extension*, where activities and states are added.

The remainder of this paper is structured as follows: Sect. 2 gives a brief introduction of behavior diagrams and the specialization of behavior diagrams, Sect. 3 gives an overview of heterogeneities between the views and an overview of the integration process, Sect. 4 describes each step of the integration process in detail, and Sect. 5 concludes the work.

2 Object/Behavior Diagrams

In this section, we briefly introduce *Object/Behavior Diagrams (OBD)*, which have been originally presented as a graphical notation for the object-oriented design of databases [7] and have been later extended for the modeling of business processes [3]. We omit the description of object diagrams and concentrate on behavior diagrams with arc-labels and their specialization; for details the reader is referred to [10,11].

2.1 Behavior Diagrams

Behavior diagrams are based on Petri nets and consist of activities, states, and arcs. Activities correspond to transitions in Petri nets and represent work performed with objects, states correspond to places in Petri nets and show where

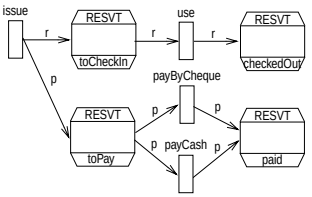


Fig. 1. Behavior diagram of RESVT

an object actually resides. Each instance of an object type is represented by a unique token, the object identifier, which may reside in one or several states. An activity may be invoked on an object if all prestates are marked by this object. When the execution is completed the object is inserted into all poststates of the activity. In difference to Petri nets, we assume that activities may take some time. During the execution of an activity on some object, the object resides in an implicit activity state named after the activity.

Example 1. Fig. 1 shows a behavior diagram of object type RESVT. Activities are depicted by rectangles, and states are depicted by rectangles with a trapezium at the top and the bottom. Ignore the labels associated with arcs for the moment. Activity issue creates a new reservation object. After completion of this activity, the object resides in states toCheckIn and toPay, where activities use and payByCheque or payCash may be started.

Behavior diagrams may be *labeled*. The idea of labeling is taken from the processing of business processes by paper work where different actors work on different carbon copies of a business form. In this analogy, a label corresponds to a particular carbon copy. The labels of an arc (state, or activity) indicate which copies of a form flow along an arc, (reside in some state, or are processed by an activity, resp.). Notice, however, that *there exist no actual copies of an object*, but several activities and states may refer at the same time to the same object by its object identifier. Labels have been introduced in [11] to facilitate the check whether a behavior diagram constitutes a consistent refinement of another behavior diagram. There, labels are used to ensure that if an object leaves one state of a subdiagram (that constitutes the refinement of an abstract state), it leaves the subdiagram entirely.

Example 2. The business process shown in Fig. 1 uses two labels p and r corresponding to the payment and the registration copy of the reservation form.

Formally, *labeled behavior diagrams* are defined in Def. 1. For a formal definition of *unlabeled* behavior diagrams see [10].

Definition 1. *A* labeled behavior diagram *(LBD) B of an object-type, where $B = (S, T, F, L, l)$, consists of a set of states $S \neq \emptyset$, a set of activities $T \neq \emptyset$, $S \cap T = \emptyset$, a set of arcs $F \subseteq (S \times T) \cup (T \times S)$, such that $\forall t \in T : (\exists s \in S : (s,t) \in F) \wedge (\exists s \in S : (t,s) \in F)$ and $\forall s \in S : (\exists t \in T : (s,t) \in F) \vee (\exists t \in T : (t,s) \in F)$. L is a set of labels. The labeling function $l : F \to 2^L \setminus \{\emptyset\}$ (2^X denotes the power*

set of X) assigns a non-empty set of labels to each arc in F. States, activities, and labels are refered to as elements. *There is a set of* initial states $A \subset S$ *such that* $\nexists(\_, s) \in F : s \in A$ *and there exists a set of* final states $\Omega \subset S$ *such that* $\nexists(s, \_) \in F : s \in \Omega$. *Initial states are usually omitted in the graphical representation.*

Labeled behavior diagrams must fulfill the following labeling properties: *(1) label preservation* (for each activity, the union of the labels of its incoming arcs is equal to the union of the labels of its outgoing arcs), *(2) unique label distribution* (all incoming arcs as well as all outgoing arcs of an activity have disjoint sets of labels), and *(3) common label distribution* (for each state, all its incident arcs carry the same labels).

States and activities are labeled, too, where the set of labels of a state or activity, denoted as λ, is the union of the set of labels of its incident arcs.

Each label appears in exactly one initial state, i.e., $(\bigcup_{\alpha \in A} \lambda(\alpha)) = L$ and $\forall s_1, s_2 \in A : s_1 \neq s_2 \Rightarrow \lambda(s_1) \cap \lambda(s_2) = \emptyset$. An initial state represents the period of time when a certain label has not yet been created for an object. If an object identifier resides only in initial states, it does not yet refer to an object.

At any time, an object resides in a non-empty set of states, its *life cycle state*.

Definition 2. *A* life cycle state *(LCS)* σ *of an object is a subset of* $(S \cup T) \times L$. *The initial LCS is* $\sigma_A = \{(\alpha, x) | \alpha \in A \wedge x \in \lambda(\alpha)\}$. *An LCS* σ *is* final *if* $\forall (e, x) \in \sigma : e \in \Omega$. Note: *For* unlabeled *behavior diagrams, an LCS is defined as* $\sigma \subset (S \cup T)$.

As the execution of activities may take some time and an object resides in an activity state during execution, we distinguish *start* and *completion* of an activity and define the change of the LCS of an object as follows (we use the notation ${}^\bullet t$ and $t^\bullet$ for the set of pre- and poststates of t, respectively).

Definition 3. *An activity* $t \in T$ *may be started on an LCS* σ, *if* $\forall s \in {}^\bullet t, x \in l(s,t) : (s,x) \in \sigma$, *and its start yields* $\sigma' = (\sigma \setminus \{(s,x) | s \in {}^\bullet t \wedge x \in l(s,t)\}) \cup \{(t,x) | \exists s \in {}^\bullet t : x \in l(s,t)\}$. *An activity t may be completed on an LCS* σ, *if* $\exists x \in L : (t,x) \in \sigma$, *and its completion yields* $\sigma' = (\sigma \setminus \{(t,x) | \exists s \in t^\bullet : x \in l(t,s)\}) \cup \{(s,x) | s \in t^\bullet \wedge x \in l(t,s)\}$.

A *life cycle occurrence* is defined as the sequence of life cycle states of a certain object:

Definition 4. *A* life cycle occurrence (LCO) γ *of an object is a sequence of LCSs* $\gamma = [\sigma_1, \ldots, \sigma_n]$, *such that* $\sigma_1 = \sigma_A$, *and for* $i = 1, \ldots, n-1$ *either* $\sigma_i = \sigma_{i+1}$, *or* $\exists t \in T$ *such that either t can be started on* σ_i *and the start of t yields* σ_{i+1} *or* $\exists x \in L : (t,x) \in \sigma_i$ *and the completion of t yields* σ_{i+1}.

Example 3. Consider the behavior diagram shown in Fig. 1. A possible life cycle occurrence of a reservation object is [{(α, p), (α, r)}, {(issue, p), (issue, r)}, {(toPay, p), (toCheckIn, r)}, {(toPay, p), (use, r)}, {(toPay, p), (checkedOut, r)}, {(payCash, p), (checkedOut, r)}, {(paid, p), (checkedOut, r)}].

2.2 Specialization of Behavior Diagrams

The behavior diagram of an object type may be specialized in two ways: by *refinement*, i.e., by decomposing states and activities into subdiagrams and labels into sublabels, or by *extension*, i.e., by adding states, activities, and labels. *Observation consistency* as a correctness criterion for specialization guarantees that any life cycle occurrence of a subtype is observable as a life cycle occurrence of the supertype if extended elements are ignored and refined elements are considered unrefined. Observation consistency allows *"parallel extension"* but not *"alternative extension"*, i.e., an activity that is added in the behavior diagram of a subtype may not consume from or produce into a state that the subtype inherits from the behavior diagram of the supertype [10]. Observation consistent extension requires only partial inheritance of activities and states: "alternatives" modeled in the behavior diagram of a supertype may be omitted in the behavior diagram of a subtype (cf. [10]). [1]

We use a total specialization function $h : S' \cup T' \cup L' \rightarrow S \cup T \cup L \cup \{\varepsilon\}$ to represent the correspondences between a more special behavior diagram $B' = (S', T', F', L', l')$ and a behavior diagram $B = (S, T, F, L, l)$. Inheritance without change, refinement, extension, and elimination are expressed by h as follows: If an element e is not changed, then $\exists e' \in S' \cup T' \cup L' : h(e') = e \wedge \forall e'' \in S' \cup T' \cup L', e'' \neq e' : h(e'') \neq h(e')$. If an element e in B is refined to a set of elements E ($|E| > 1$), then $\forall e' \in E : h(e') = e$. If a set of elements E is added in B', then $\forall e \in E : h(e) = \varepsilon$. If a set of states and activities $E \subseteq S \cup T$ is removed from B in B', then $\forall e \in E \,/\, e \not\exists \in S' \cup T' \cup L' : h(e') = e$.

For the definition of *observation consistent specialization*, we need to define the *generalization of a life cycle state and a life cycle occurrence.*

Definition 5. *A* generalization of a life cycle state σ' *of an LBD* B' *of object type* O' *to object type* O *with LBD* B*, denoted as* $\sigma'/_O$*, is defined as* $\sigma'/_O \subseteq (S \cup T) \times L$*, where* $\forall e \in S \cup T, x \in L : ((e, x) \in \sigma'/_O \Leftrightarrow \exists e' \in S' \cup T', x' \in L' : h(e') = e \wedge h(x') = x \wedge (e', x') \in \sigma')$.

Definition 6. *A* generalization of a life cycle occurrence $\gamma' = [\sigma'_1, \ldots, \sigma'_n]$ *of an LBD* B' *of object type* O' *to object type* O *is defined as* $\gamma'/_O = [\sigma'_1/_O, \ldots, \sigma'_n/_O]$.

Definition 7. *An LBD* B' *of object type* O' *is an* observation consistent specialization *of an LBD* B *of object type* O *if and only if for any possible LCO* γ' *of* B' *holds:* $\gamma'/_O$ *is an LCO of* B.

[1] Omitting states and activities of a supertype in a subtype seems to be counter-intuitive at first. Yet, a more in depth analysis (cf. [10]) reveals that partial inheritance is coherent with a co-variant specialization of method preconditions followed in conceptual specification languages.

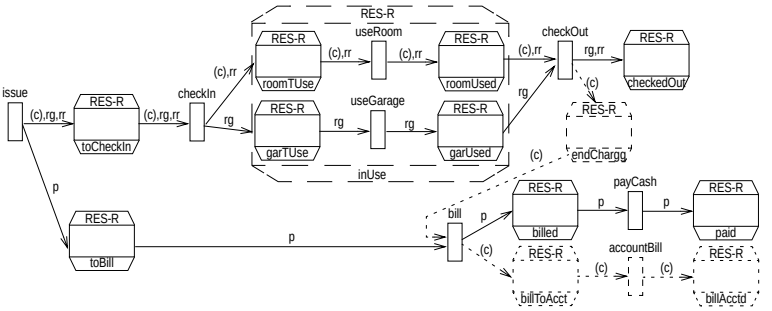


Fig. 2. Business process of RES-R

Example 4. The LBD of RES-R shown in Fig. 2 is an observation consistent specialization of the LBD of RESVT in Fig. 1. To avoid overloading of Fig. 2, the unrefined elements inherited from RESVT are not depicted. The symbol with dashed borders, the brackets around label c, and the fact that some symbols are depicted with dotted lines are explained in the next section.

Activity use has been refined to a subnet consisting of activities checkIn, useRoom, useGarage, and checkOut and several states. State toPay has been refined to toBill, bill, and billed. Label r has been decomposed into sublabels rr (registration for a room) and rg (registration for a parking lot in the garage). Activity accountBill, states endChargg, billToAcct, and billAcctd, as well as label c have been added by extension. Activity payByCheque has been omitted.

A set of sufficient and necessary rules has been introduced in [11] to check for *observation consistent extension* and *refinement* of LBDs. These rules can be applied to check for *observation consistent specialization* of LBDs, too, if specialization is considered as a concatenation of refinement and extension.

3 Integration of Views of Business Processes

In this section, we describe how to integrate *two* views of a business process. To integrate more than two views, a *binary integration strategy* [1] may be used, which integrates each view with an intermediate integrated schema. The views may differ with respect to several kinds of heterogeneities. We will identify these heterogeneities first; then we will describe the integration process.

3.1 Heterogeneities

Some of the heterogeneities between different views of business processes resemble the heterogeneities identified for federated databases [12]. Others, especially the second and last one described below are unique for business processes: *(1) Model conflicts:* The views may be represented in different models (e.g., statecharts vs. Petri nets). *(2) Inclusion of workflow aspects:* The views may represent workflows rather than business processes (cf. discussion in the introduction). *(3) Missing states and labels:* States and labels that are modeled in

one view may not be modeled explicitly in the other view, but may exist there only implicitly. *(4) Naming conflicts:* Corresponding elements may carry different names *(synonyms)* or different elements may carry the same names *(homonyms)*. *(5) Granularity conflicts:* A single state or activity in one view may correspond to a subnet consisting of several activities and states in the other view, or two subnets may correspond to each other. *(6) View-specific alternatives:* One view may include an alternative that is not modeled in the other view; i.e., an activity that consumes from or produces into a state that has a corresponding state in the other view does not have a corresponding activity in the other view. *(7) View-specific aspects:* One view may consider aspects represented by labels that are not modeled in the other view, i.e., a label in one view does not correspond to any label in the other view. *(8) Different time periods:* The views may represent different periods of the lifetime of business objects. This heterogeneity may be considered as a special kind of granularity conflict, in which the initial or final states of one view correspond to subnets in the other view.

3.2 Overview of the Integration Process

The integrated business process has to be constructed from the given views in a way such that the processing of business objects according to the integrated business process may be observed as correct processing of the business objects according to each given view, i.e., the integrated business process must be an observation consistent specialization of each given view.

As views do not administrate information of their own about business objects, all information that is visible in the view must be derived from the integrated business process. Therefore, the integrated business process must include all information specified in either one of the views, except view-specific alternatives, which must be omitted to provide for an observation consistent specialization, since performing a view-specific alternative on a business object could not be observed in the view that does not include this alternative.

If corresponding elements are defined in the views at different levels of granularity, the most refined definition is taken into the integrated schema to provide the most detailed information possible.

Before the views can be integrated according to the rules of observation consistent specialization, some preparatory steps (steps 1–3) are performed on the original views, thereby yielding *enriched views*. Then, the integrated business process is defined by *refinement* and subsequent *extension* of the enriched views. Some steps are comparable to the steps followed in several approaches of database integration (cf. [2,13]). In the remainder of this section, we give an overview of the integration process. The steps of the integration process will be treated in detail together with examples in the next section.

1. *Schema translation:* If the given views are modeled using different models, they are translated to a canonical model, thereby resolving *model conflicts*.
2. *Extraction of business processes from workflows:* Heterogeneities due to *inclusion of workflow aspects* are resolved in this phase as only elements concerning the business process are considered.

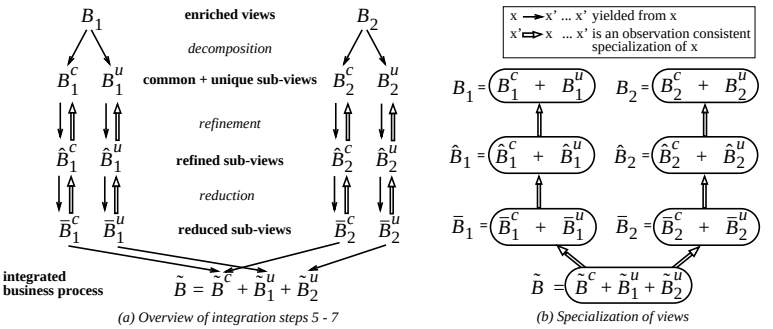


Fig. 3. Integration process by observation consistent specialization

3. *Schema enrichment:* Information that exists only implicitly in a given view is made explicit. Thereby, *missing states and labels* are defined.
   Labels should help the system integrator to determine common aspects of the given views and to check for correct refinements of activities and states. Labels can be determined according to the analogy with paper forms introduced in the previous section. Notice that this analogy should only be a hint to find labels as no real paper copies have to exist. A redundant state has to be inserted if it represents the processing state of a certain label but has not been modeled so far in the view.
4. *Identification of correspondences:* Correspondences between activities, states, and labels of the enriched views are determined. Elements of a view that have corresponding elements in the other view are called *common,* all other elements are called *unique.*
5. *View decomposition:* Common elements and view-specific aspects are integrated in two separate steps. To determine the sub-views to be considered in these steps, each *enriched view* B_i (for $i \in \{1, 2\}$) is decomposed into (1) a *common sub-view* B_i^c, which contains the restriction of the view to all common labels and all activities, states, and arcs that are labeled with common labels, and (2) a *unique sub-view* B_i^u, which consists of all unique labels and all activities, states, and arcs that are labeled with unique labels (cf. Fig. 3 (a)). These sub-views may be overlapping, as an activity, state or arc may be labeled with common *and* unique labels. Notice that by decomposing an enriched view into a common and a unique sub-view no information is added or removed.
6. *Integration of common elements:* The common sub-views B_1^c and B_2^c are integrated to the *common view* $\tilde{B}^c$. Thereby, *granularity conflicts*, heterogeneities due to *view-specific alternatives,* and *naming conflicts* are resolved. To resolve granularity conflicts, the common sub-views B_1^c and B_2^c are refined to a common level of granularity according to the rules of observation consistent refinement (cf. [11]) yielding *common refined sub-views* $\hat{B}_1^c$ and $\hat{B}_2^c$. To resolve heterogeneities due to view-specific alternatives, alternatives with common labels in one view that are not modeled in the other view are omitted from $\hat{B}_1^c$ and $\hat{B}_2^c$, whereby the rules of observation consistent

extension are obeyed because of the rule of *partial inheritance* (cf. Sect. 2.2). The resulting reduced sub-views $\bar{B}_1^c$ and $\bar{B}_2^c$ are identical up to renaming. To resolve naming conflicts, a common view $\tilde{B}^c$ that is isomorphic to $\bar{B}_1^c$ and $\bar{B}_2^c$ is defined, where the elements of $\tilde{B}^c$ have different "object identities" than their counterpart elements of $\bar{B}_1^c$ and $\bar{B}_2^c$ and possibly different names.

7. *Integration of view-specific aspects:* For each *unique sub-view* B_i^u, activities and states that are included in the unique sub-view B_i^u as well as in the common sub-view B_i^c and that are refined in $\hat{B}_i^c$, are refined in B_i^u, too, yielding a *refined unique sub-view* $\hat{B}_i^u$.
   Then for each $\hat{B}_i^u$, view-specific alternatives that have been omitted in the reduced common sub-view $\bar{B}_i^c$ are omitted from the refined unique sub-view $\hat{B}_i^u$, too, yielding a *reduced unique sub-view* $\bar{B}_i^u$. Naming conflicts (homonyms) between $\bar{B}_1^u$ and $\bar{B}_2^u$ are resolved — as described above for the common sub-views — yielding $\tilde{B}_1^u$ and $\tilde{B}_2^u$. Overlaps between B_i^c and B_i^u are taken into account in the definition of the object-identities of $\tilde{B}_i^u$ in that an element in $\tilde{B}^c$ and an element in $\tilde{B}_i^u$ that have the same counterpart in B_i receive the same "object identity".
   Heterogeneities due to *view-specific aspects* are resolved in that the integrated business process $\tilde{B}$ is defined as composition $\tilde{B} := \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ (cf. Def. 8).

Definition 8. *The* composition of behavior diagrams $B_a = (S_a, T_a, F_a, L_a, l_a)$ *and* $B_b = (S_a, T_b, F_b, L_b, l_b)$ *(denoted* $B_a + B_b$*) yields a behavior diagram* $B_c = (S_c, T_c, F_c, L_c, l_c)$*, where* $S_c = S_a \cup S_b, T_c = T_a \cup T_b, F_c = F_a \cup F_b, L_c = L_a \cup L_b, l_c = \{(f, X) | f \in F_c \wedge X = l_a(f) \cup l_b(f)\}$.

As the common view B_i^c and the unique view B_i^u are refined the same way and as $\hat{B}_i^c$ is an observation consistent refinement of B_i^c, and as $\hat{B}_i^u$ is an observation consistent refinement of B_i^u, it holds that $\hat{B}_i = \hat{B}_i^c + \hat{B}_i^u$ is an observation consistent refinement of B_i (cf. Fig. 3 (b)). Since $\bar{B}_i^c$ is an observation consistent extension of $\hat{B}_i^c$, and since $\bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i^u$, it holds that $\bar{B}_i = \bar{B}_i^c + \bar{B}_i^u$ is an observation consistent extension of $\hat{B}_i$. As the common sub-views $\bar{B}_1^c$ and $\bar{B}_2^c$ are isomorphic up to renaming, as the sets of labels of the common sub-views are disjoint with the set of labels of each unique subview $\bar{B}_1^u$ and $\bar{B}_2^u$, and as the labels of the unique sub-views are disjoint, too, it holds that the integrated business process $\tilde{B} = \tilde{B}^c + \tilde{B}_1^u + \tilde{B}_2^u$ is an observation consistent specialization of B_1 and B_2.

4 The Steps of the Integration Process

In this section, we will present the steps of integration process in more detail, illustrated by an example described below. Notice that, although the steps are described sequentially, there may be several iterations, especially between steps 3 and 7, as lacking redundant states may be recognized only during specialization.

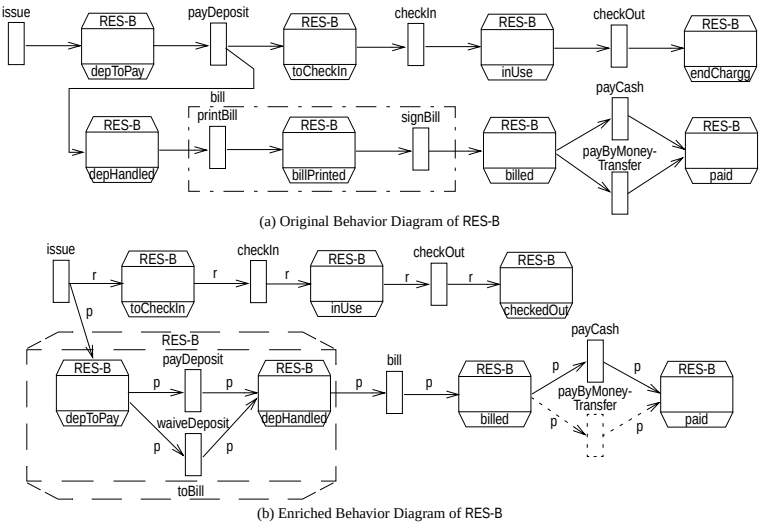


Fig. 4. Behavior Diagrams of RES-B

Example 5. Two views of treating room reservations have to be integrated to a single business process for a hotel. The views consider handling of room reservations from the viewpoints of the reception and of the booking department (cf. Figs. 2 and 4, respectively). The integration steps *extraction of business processes from workflows* and *schema enrichment* will be explained only with RES-B.

4.1 Schema Translation

The translation step is out of the scope of this paper. Herein, we assume that the given view schemas have been translated to unlabeled Behavior Diagrams.

4.2 Extraction of Business Processes from Workflows

The business process is extracted from each view schema that represents a workflow rather than a business process: (a) order dependencies between activities that reflect the intended flow of work rather than the business process itself are omitted; (b) alternative activities that could be executed but are not supported in the considered workflow are included in the view; (c) exact instructions on how a certain unit of work has to be performed by an agent are replaced by the overall business process activity. By omitting restrictions due to internal business rules, contradictions between the views disappear.

Example 6. Consider the view in Fig. 4 (a). Due to an internal business rule a deposit has to be paid before check-in, although, by law, a deposit is not necessary and — in case that it is required — need not necessarily be charged *before* check-in. Further, activities printBill and signBill and state billPrinted, which represent an internal instruction on how billing is performed by a clerk are replaced by the overall activity bill.

4.3 Schema Enrichment

Information that is not modeled explicitly, but exists only implicitly, is made explicit. Particularly, labels are defined and redundant states may be inserted, i.e., an unlabeled behavior diagram $\breve{B}_i = (\breve{S}_i, \breve{T}_i, \breve{F}_i)$ is transformed into a labeled behavior diagram $B_i = (S_i, T_i, F_i, L_i, l_i)$, where $\breve{S}_i \subseteq S_i, \breve{T}_i = T_i, \breve{F}_i \subseteq F_i$.

Redundant states must not restrict the possible sequences of activity invocations, i.e., the following condition must hold: *For any LCO* $\breve{\gamma} = [\breve{\sigma}_1, \ldots, \breve{\sigma}_n]$ *of* $\breve{B}_i$*, a labeled LCO* $\gamma = [\sigma_1, \ldots, \sigma_n]$ *of* B_i *has to exist such that for* $1 \leq j \leq n$ *it holds that:* $\forall \breve{s} \in \breve{\sigma}_j \exists (\breve{s}, x) \in \sigma_j$.

Example 7. In view RES-B (cf. Fig. 4), labels p and r (representing the aspects *payment* and *registration,* respectively) are introduced, yielding the enriched view shown in Fig. 4 (b).

Assume that the *unlabeled* view RES-R (cf. Fig. 2) did not contain states checkedOut and toBill. Then, these redundant states are inserted during schema enrichment as they represent processing states of labels rg, rr, and p.

4.4 Identification of Correspondences

An activity, state, or label in one view may correspond to an activitiy, state, or label in the other view, either at the same or at different levels of granularity. Thus, we distinguish the following types of correspondences: *(1) Equivalence:* Two activities, states, or labels are equivalent if they correspond to each other. *(2) Inclusion:* A state or activity in one view corresponds to a subnet of activities and states in the other view. Similarly, a single label in one view may correspond to a set of labels in the other view. *(3) No correspondence:* An element in one view corresponds to no element in the other view.

Example 8. In our example, several activities, states, and labels are considered equivalent between the views (cf. Figs. 2 and 4 (b)). For better readability, they carry the same names. E.g., activities issue, states checkedOut, and labels p are equivalent. There exists an inclusion relationship between state inUse in view RES-B and the subnet consisting of roomTUse, garToUse, roomUsed, garUsed, useRoom, and useGarage in RES-R. Further, label r in RES-B corresponds to the set of labels rg and rr in RES-R. Common activities and states are depicted with solid borders, common labels are depicted without brackets, unique activities and states are depicted with dotted borders, unique labels are enclosed in brackets. Subnets in one view involved in an inclusion with a single state e in the other view are visualized by a symbol with dashed borders carrying the name of e.

Subnet correspondences, i.e., the fact that two subnets correspond to each other, cannot occur if object life-cycles of two object types modeling the same set of objects are integrated. A subnet correspondence means that an abstract activity or an abstract state is refined differently in two views but an object can be treated only in one way in the integrated schema. Supposed subnet correspondences indicate design errors such as: (1) At least one subnet does

$\Theta_S^E \subseteq S_1 \times S_2$	$\Theta_T^E \subseteq T_1 \times T_2$	$\Theta_L^E \subseteq L_1 \times L_2$
$\Theta_S^{I_1} \subseteq S_1 \times 2^{S_2 \cup T_2}$	$\Theta_T^{I_1} \subseteq T_1 \times 2^{S_2 \cup T_2}$	$\Theta_L^{I_1} \subseteq L_1 \times 2^{L_2}$
$\Theta_S^{I_2} \subseteq 2^{S_1 \cup T_1} \times S_2$	$\Theta_T^{I_2} \subseteq 2^{S_1 \cup T_1} \times T_2$	$\Theta_L^{I_2} \subseteq 2^{L_1} \times L_2$
$\Theta_S := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_S^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_S^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_S^{I_2}\}$		
$\Theta_T := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_T^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_T^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_T^{I_2}\}$		
$\Theta_L := \{(\{e_1\},\{e_2\}) \mid (e_1,e_2) \in \Theta_L^E\} \cup$		
$\{(\{e_1\},E_2) \mid (e_1,E_2) \in \Theta_L^{I_1}\} \cup \{(E_1,\{e_2\}) \mid (E_1,e_2) \in \Theta_L^{I_2}\}$		

Table 1. Correspondence relations

not model a pure business process, but considers internal business rules. (2) The subnets consider different aspects (i.e., different labels), and thus, they do not correspond to each other at all. (3) A single element in one subnet corresponds to a single element or a set of elements in the other subnet, i.e., correspondences have not been detected at the finest level of granularity.

We require that correspondences are not overlapping, i.e., that each element e in a view corresponds to at most one element or one subnet E in the other view. Overlapping correspondences may occur if activities and states within the given views have been insufficiently refined such that there are aspects that are defined at an abstract level in both views. In such a case, the views have to be refined such that for each element there is a most-refined representation in at least one view.

For the remainder of the integration process, we define the correspondence relation $\Theta := \Theta_S \cup \Theta_T \cup \Theta_L$ for the enriched views B_1 and B_2, where $B_i = (S_i, T_i, F_i, L_i, l_i)$, for $i \in \{1,2\}$, and where Θ_S, Θ_T, and Θ_L represent correspondences between states, activities, and labels, respectively (cf. Table 1). For each of these correspondence relations, we distinguish between *(1) equivalences* Θ^E (i.e., two basic elements correspond to each other), *(2) left inclusions* Θ^{I_1} (i.e., a single element in B_1 corresponds to a subnet in B_2), and *(3) right inclusions* Θ^{I_2} (i.e., a single element in B_2 corresponds to a subnet in B_1).

As overlapping correspondences are not allowed, no element may appear in more than one correspondence relation, i.e., $\forall (E_1', E_2') \in \Theta, (E_1'', E_2'') \in \Theta : (E_1', E_2') \neq (E_1'', E_2'') \Rightarrow (E_1' \cap E_1'' = \emptyset \wedge E_2' \cap E_2'' = \emptyset)$.

Further, we require that all common activities and states are labeled with at least one common label, i.e., for $i \in \{1,2\}$, it holds that $\forall (E_1, E_2) \in \Theta_S \cup \Theta_T, e \in E_i \; \exists (X_1, X_2) \in \Theta_L : \lambda_i(e) \cap X_i \neq \emptyset$, where $\lambda_i : S_i \cup T_i \rightarrow L_i$ is the function describing the labeling of elements in B_i.

Example 9. View RES-B is considered as B_1, view RES-R is considered as B_2. There are several equivalences, e.g., $(issue, issue) \in \Theta_T^E$, and inclusions, e.g., the left inclusion $(r, \{rg, rr\}) \in \Theta_L^{I_1}$, and the right inclusion $(\{depToPay, payDeposit, waiveDeposit, depHandled\}, toBill) \in \Theta_S^{I_2}$.

$B_i^c = (S_i^c, T_i^c, F_i^c, L_i^c, l_i^c)$, where	$B_i^u = (S_i^u, T_i^u, F_i^u, L_i^u, l_i^u)$, where
$L_i^c = \{x \in L_i \mid \exists (E_1, E_2) \in \Theta_L : x \in E_i\}$	$L_i^u = \{x \in L_i \mid / (\nexists E_1, E_2) \in \Theta_L : x \in E_i\}$
$S_i^c = \{s \in S_i \mid \lambda_i(s) \cap L_i^c \neq \emptyset\}$	$S_i^u = \{s \in S_i \mid \lambda_i(s) \cap L_i^u \neq \emptyset\}$
$T_i^c = \{t \in T_i \mid \lambda_i(t) \cap L_i^c \neq \emptyset\}$	$T_i^u = \{t \in T_i \mid \lambda_i(t) \cap L_i^u \neq \emptyset\}$
$F_i^c = F_i \cap ((S_i^c \cup T_i^c) \times (S_i^c \cup T_i^c))$	$F_i^u = F_i \cap ((S_i^u \cup T_i^u) \times (S_i^u \cup T_i^u))$
$l_i^c = \{(f, X) \mid f \in F_i^c \wedge X = l_i(f) \cap L_i^c\}$	$l_i^u = \{(f, X) \mid f \in F_i^u \wedge X = l_i(f) \cap L_i^u\}$

Table 2. Definition of the common and unique sub-views

4.5 View Decomposition

In this step, for each enriched view B_i, the *common sub-view* B_i^c and the *unique sub-view* B_i^u are defined: The *common sub-view* consists of all *common labels* as well as activities, states, and arcs that are labeled with at least one common label; the *unique sub-view* consists of all *unique labels* as well as activities, states, and arcs that are labeled with at least one unique label. Formally, B_i^c and B_i^u are defined in Table 2.

Example 10. In the enriched view RES-R (cf. Fig. 2), labels p, rg, and rr and all states, activities, and arcs labeled with at least one of these labels are taken into the common sub-view. The unique sub-view of RES-R consists of label c as well as all states, activities, and arcs that are labeled with c.

4.6 Integration of Common Elements

In this step, the *common view* $\tilde{B}^c$ is defined based on the *common sub-views* B_1^c and B_2^c and the determined correspondences. We will omit definition of the intermediate views $\hat{B}_i^c$ and $\bar{B}_i^c$, but define the common view $\tilde{B}^c = (\tilde{S}^c, \tilde{T}^c, \tilde{F}^c, \tilde{L}^c, \tilde{l}^c)$ immediately from the common sub-views B_1^c and B_2^c. The introduced specialization function h_i^c may be decomposed (cf. Sect. 2.2) to yield the intermediate views.

In $\tilde{B}^c$, a state, an activity, or a label is defined for each common element; we define each element in $\tilde{B}^c$ by an ordered pair that represents the elements of both views that correspond to this element. The refinement function $h_i^c : (\tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c) \to (S_i^c \cup T_i^c \cup L_i^c)$ is defined for each B_i^c (cf. Tab. 3).

Example 11. The common view of RES-R and RES-B (cf. Figs. 2 and 4 (b)) is defined as shown in Fig. 5, except that the activities and states with dotted borders and the labels put in brackets are not yet included. For better readability, the elements in the common view carry the same name as the corresponding elements in the enriched views RES-B and RES-R.

The common view $\tilde{B}^c$ is correct only if it is an LBD according to Def. 1 and if it is an observation consistent specialization of B_1^c and B_2^c. If these criteria are not fulfilled, the determined correspondences are not correct or the given enriched views are contradictory. An error occurs, for example, if elements that are considered to be corresponding do not behave the same way, e.g., there is an activity and a state in one view that are considered equivalent to an activity

$$
\begin{array}{l}
\tilde{S}^c := \{(e_1,e_2)|(e_1,e_2) \in \Theta_S^E\} \cup \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in S_2^c \cap E_2\} \cup \\
\qquad \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in S_1^c \cap E_1\} \\
\tilde{T}^c := \{(e_1,e_2)|(e_1,e_2) \in \Theta_T^E\} \cup \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_S^{I_1} \cup \Theta_T^{I_1} : e_2 \in T_2^c \cap E_2\} \cup \\
\qquad \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_S^{I_2} \cup \Theta_T^{I_2} : e_1 \in T_1^c \cap E_1\} \\
\tilde{L}^c := \{(e_1,e_2)|(e_1,e_2) \in \Theta_L^E\} \cup \{(e_1,e_2)|\exists (e_1,E_2) \in \Theta_L^{I_1} : e_2 \in E_2\} \cup \\
\qquad \{(e_1,e_2)|\exists (E_1,e_2) \in \Theta_L^{I_2} : e_1 \in E_1\} \\
\hline
h_1^c(e^c) := e_1^c \text{ and } h_2^c(e^c) := e_2^c, \text{ where } e^c = (e_1^c, e_2^c) \\
\hline
\tilde{F}^c := \{(e', e'')|e' \in \tilde{S}^c \cup \tilde{T}^c \wedge e'' \in \tilde{S}^c \cup \tilde{T}^c \wedge \\
\qquad (((h_1^c(e'), h_1^c(e'')) \in F_1^c \wedge (h_2^c(e'), h_2^c(e'')) \in F_2^c) \vee \\
\qquad ((h_1^c(e'), h_1^c(e'')) \in F_1^c \wedge h_2^c(e') = h_2^c(e'')) \vee \\
\qquad (h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'), h_2^c(e'')) \in F_2^c))\} \\
\tilde{l}^c := \{((e', e''), X)|(e', e'') \in \tilde{F}^c \wedge X = \{x \in \tilde{L}^c| \\
\qquad ((h_1^c(e'), h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'), h_1^c(e'')) \wedge \\
\qquad (h_2^c(e'), h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'), h_2^c(e'))) \vee \\
\qquad ((h_1^c(e'), h_1^c(e'')) \in F_1^c \wedge h_1^c(x) \in l_1^c(h_1^c(e'), h_1^c(e'')) \wedge h_2^c(e') = h_2^c(e'')) \vee \\
\qquad (h_1^c(e') = h_1^c(e'') \wedge (h_2^c(e'), h_2^c(e'')) \in F_2^c \wedge h_2^c(x) \in l_2^c(h_2^c(e'), h_2^c(e'')))\}\}
\end{array}
$$

Table 3. Definition of the common view

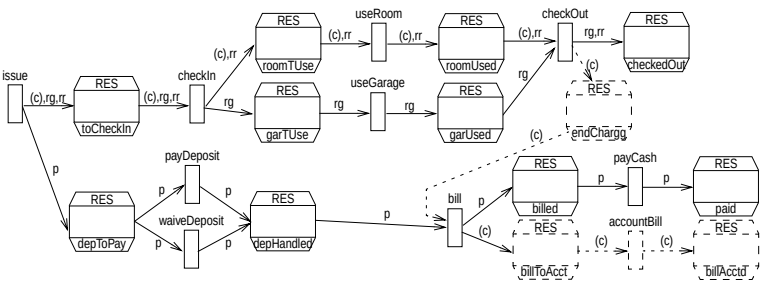


Fig. 5. Integrated business process: Behavior diagram of RES

and a state, respectively, in the other view; in one view there is an arc between the activity and the state, but in the other view, there is no arc.

Example 12. Consider the case that activity payCash has been omitted in RES-B such that only activity payByMoneyTransfer is left. This activity does not correspond to payCash in view RES-R, but state paid in RES-B corresponds to state paid in RES-R. Then, the common view would include a state named paid, which is not connected to any activity. The reason is that the views cannot be integrated if there is no common means of payment defined.

4.7 Integration of View-Specific Aspects

In the last step, the unique labels are treated to resolve heterogeneities due to *view-specific aspects*.

States and activities that are included in the common sub-view B_i^c as well as the unique sub-view B_i^u of a view B_i and that have been refined in the refined

common sub-view $\hat{B}_i^c$ have to be refined in the unique sub-view in the same way (yielding the *refined unique sub-view* $\hat{B}_i^u$). View-specific alternatives that have been omitted in the reduced common sub-view $\bar{B}_i^c$ have to be omitted in the unique sub-view, too (yielding the *reduced unique sub-view* $\bar{B}_i^u$). For all elements of the reduced unique sub-view, a new element is defined in the *unique view* $\tilde{B}_i^u$.

When an activity or a state that is labeled with a unique label is refined in the common view and, thus, in the unique view, the system integrator has to decide how the unique label is treated within the refined state or activity. This problem particularly arises if an activity or state is refined to a subnet including parallel states and activities that enforce splitting of the unique label.

The unique views $\tilde{B}_i^u$ may be defined directly from the unique sub-views B_i^u; $\tilde{B}_i^u$ has to be a correct LBD and an observation consistent specialization of B_i^u based on the specialization function $h_i^u : \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u \to S_i^u \cup T_i^u \cup L_i^u$.

The specialization of the unique sub-views must not contradict the specialization of the common sub-views. Therefore, the following conditions must hold: (1) $\forall x' \in \tilde{L}_i^u : h_i^u(x') \in L_i^u$ (i.e., labels may be refined but no new labels are introduced), (2) $\forall e \in (S_i^c \cup T_i^c) \cap (S_i^u \cup T_i^u), e' \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{S}_i^u \cup \tilde{T}_i^u : h_i^c(e') = e \Leftrightarrow h_i^u(e') = e$ (i.e., all elements that exist in the common sub-view as well as in the unique sub-view have to be refined in the same way), (3) $\forall e' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u), e'' \in (\tilde{S}^c \cup \tilde{T}^c) \cap (\tilde{S}_i^u \cup \tilde{T}_i^u) : (e', e'') \in \tilde{F}^c \Leftrightarrow (e', e'') \in \tilde{F}_i^u$ (i.e., the same set of arcs is defined for elements that exist in the common view as well as in the unique sub-view).

The integrated business process $\tilde{B} = (\tilde{S}, \tilde{T}, \tilde{F}, \tilde{L}, \tilde{l})$ is a composition of $\tilde{B}^c, \tilde{B}_1^u$, and $\tilde{B}_2^u$. It will be an observation consistent specialization of B_1 and B_2, if $\tilde{B}^c$ is an observation-consistent specialization of B_1^c and B_2^c, and for $i \in \{1, 2\}$, $\tilde{B}_i^u$ is an observation-consistent specialization of B_i^u. The specialization function $\tilde{h}_i : \tilde{S} \cup \tilde{T} \cup \tilde{L} \to S_i \cup T_i \cup L_i \cup \{\varepsilon\}$, which represents the specialization of B_i to $\tilde{B}$ is defined as follows: $\tilde{h}_i(\tilde{e}) := h_i^c(\tilde{e})$ if $\tilde{e} \in \tilde{S}^c \cup \tilde{T}^c \cup \tilde{L}^c$, $\tilde{h}_i(\tilde{e}) := h_i^u(\tilde{e})$ if $\tilde{e} \in \tilde{S}_i^u \cup \tilde{T}_i^u \cup \tilde{L}_i^u$, and $\tilde{h}_i(\tilde{e}) := \varepsilon$ else.

Example 13. The unique label c of RES-R is taken into the integrated business process RES (cf. Fig. 5).

5 Conclusion

In this paper, we introduced an approach to design a business process for an organization based on views defined by sub-units of the organization. The problem corresponds to the definition of a conceptual database schema based on external views. In this work, we treated the integration of the overall *behavior* of business objects, i.e., the integration of object life-cycles.

Different from the integration of local schemas in federated database systems, where data is shared in the local databases and data in the integrated view is derived, the integration of object life-cycles we have presented assumes that objects are instances of the integrated object type and that views represent derived information on the treatment of the business objects, possibly, restricted to a subset of aspects and at a coarser level of granularity.

We used the criterion of *observation consistent specialization,* which has been introduced for the top-down design of object-oriented databases, to check whether some business process is a correct integration of several views. Different to top-down specialization, integration requires to determine correspondences between elements of views of object life-cycles and to develop an object life-cycle that is a specialization of each given view.

References

1. C. Batini, M. Lenzerini, and S. B. Navathe. A Comparative Analysis of Methodologies for Database Schema Integration. *ACM Computing Surveys*, 18(4):323–364, December 1986. 38
2. E. Bertino and A. Illarramendi. The Integration of Heterogeneous Data Management Systems: Approaches Based on the Object-Oriented Paradigm. In [4]. 39
3. P. Bichler, G. Preuner, and M. Schrefl. Workflow Transparency. In *Proc. CAiSE'97*, Springer LNCS 1250, 1997. 33, 34
4. O. Bukhres and A. Elmagarmid. *Object-Oriented Multidatabase Systems: A Solution for Advanced Applications*. Prentice-Hall, 1996. 33, 48
5. L. Ekenberg and P. Johannesson. A Formal Basis for Dynamic Schema Integration. In *Conceptual Modeling - ER '96*, Springer LNCS 1157, 1996. 33, 34
6. H. Frank and J. Eder. Integration of Behaviour Models. In *Proc. ER '97 Workshop on Behavioral Models and Design Tranformations*, 1997. 33, 34
7. G. Kappel and M. Schrefl. Object/Behavior Diagrams. In *Proc. ICDE'91*, 1991. 34, 34
8. J. L. Peterson. Petri nets. *ACM Computing Surveys*, pages 223–252, 1977. 34
9. G. Preuner and M. Schrefl. Observation Consistent Integration of Business Processes. In *Proc. Australian Database Conference (ADC)*, Springer, 1998. 34, 34
10. M. Schrefl and M. Stumptner. Behavior Consistent Extension of Object Life Cycles. In *Proc. OO-ER '95*, Springer LNCS 1021, 1995. 34, 34, 35, 37, 37, 37
11. M. Schrefl and M. Stumptner. Behavior Consistent Refinement of Object Life Cycles. In *Proc. ER '97*, Springer LNCS 1331, 1997. 34, 34, 35, 38, 40
12. A. Sheth and V. Kashyap. So Far (Schematically) yet So Near (Semantically). In *Proc. DS-5 Semantics of Interoperable Database Systems*, 1992. 38
13. A. Sheth and J. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990. 33, 39
14. C. Thieme and A. Siebes. Guiding Schema Integration by Behavioural Information. *Information Systems*, 20(4):305–316, 1995. 33
15. W. M. P. van der Aalst and T. Basten. Life-Cycle Inheritance — A Petri-Net-Based Approach. In *Proc. PN '97*, Springer LNCS 1248, 1997. 34
16. M. Vermeer and P. Apers. Behaviour specification in database interoperation. In *Proc. CAiSE '97*, Springer LNCS 1250, 1997. 33

Efficient Mining of Association Rules in Large Dynamic Databases

Edward Omiecinski and Ashok Savasere

College of Computing, Georgia Institute of Technology, Atlanta GA 30332, USA

Abstract. Mining for association rules between items in a large database of sales transactions is an important database mining problem. However, the algorithms previously reported in the literature apply only to static databases. That is, when more transactions are added, the mining process must start all over again, without taking advantage of the previous execution and results of the mining algorithm. In this paper we present an efficient algorithm for mining association rules within the context of a dynamic database, (i.e., a database where transactions can be added). It is an extension of our *Partition* algorithm which was shown to reduce the I/O overhead significantly as well as to lower the CPU overhead for most cases when compared with the performance of one of the best existing association mining algorithms.

1 Introduction

Increasingly, business organizations are depending on sophisticated decision-making information to maintain their competitiveness in today's demanding and fast changing marketplace. Inferring valuable high-level information based on large volumes of routine business data is becoming critical for making sound business decisions. For example, customer buying patterns and preferences, sales trends, etc, can be learned by analyzing point-of-sales data at supermarkets. Discovering these new nuggets of knowledge is the intent of database mining.

One of the main challenges in database mining is developing fast and efficient algorithms that can handle large volumes of data since most mining algorithms perform computations over the entire database, which is often very large. Besides a large amount of data, another characteristic of most database systems is that they are dynamic (i.e., data can be added and deleted). This poses a potential dilemma for data mining. Since the database will never contain all the data, the data mining process can be performed either on the recent data or on all the data accumulated thus far. In this paper our focus is on applications that need the entire database mined and hence we concentrate on accomplishing this in an incremental fashion.

Discovering association rules between items over *basket* data was introduced in [1]. Basket data typically consists of items bought by a customer along with the date of transaction, quantity, price, etc. Such data may be collected, for example, at supermarket checkout counters. Association rules identify the set of

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 49–63, 1998.

items that are most often purchased with another set of items. For example, an association rule may state that "95% of customers who bought items A and B also bought C and D." Association rules may be used for catalog design, store layout, product placement, target marketing, etc.

Many algorithms have been discussed in the literature for discovering association rules [1,2,3,4,5]. One of the key features of all the previous algorithms is that they require multiple passes over the database. For disk resident databases, this requires reading the database completely for each pass resulting in a large number of disk I/Os. In these algorithms, the effort spent in performing just the I/O may be considerable for large databases. For example, a 1 GB database will require roughly 125,000 block reads for a single pass (for a block size of 8KB). If the algorithm takes, say, 10 passes, this results in 1,250,000 block reads. Assuming that we can take advantage of doing sequential I/O where we can read about 400 blocks per second, the time spent in just performing the I/O is approximately 1 hour. If we could reduce the number of passes to two, we reduce the I/O time to approximately 0.2 seconds. With dynamically changing databases, the problem is further compounded by the fact that when new data is added, the mining algorithm must be run again, just as if it were the first time the process was performed. For example, it would require making another 10 passes over the current database.

In a previous paper, we describe our algorithm called *Partition* [6], which is fundamentally different from all the previous algorithms in that it reads the database at most two times to generate all significant association rules. Contrast this with the previous algorithms, where the database is not only scanned multiple times but the number of scans cannot even be determined in advance. Surprisingly, the savings in I/O is not achieved at the cost of increased CPU overhead. We have also performed extensive experiments [6] and compared our algorithm with one of the best previous algorithms. Our experimental study shows that for computationally intensive cases, our algorithm performs better than the previous algorithm in terms of both CPU and I/O overhead.

In this paper we extend our *Partition* algorithm to accomodate dynamically changing databases where new transactions are inserted. We also provide experimental results showing the improved performance over our previous algorithm. We only consider our previous algorithm since it has been shown to have better performance over the best competing algorithm [2] for static databases.

The paper is organized as follows: in the next section, we give a formal description of association mining. In Section 2, we describe the problem and give an overview of the previous algorithms in section 3. In section 4, we describe our algorithm. Performance results are described in section 5. Section 6 contains our conclusions.

2 Association Mining

This section is largely based on the description of the problem in [1] and [2]. Formally, the problem can be stated as follows: Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a

set of m distinct literals called *items*. $\mathcal{D}$ is a set of variable length transactions over $\mathcal{I}$. Each transaction *contains* a set of items $i_i, i_j, \ldots, i_k \subset \mathcal{I}$. A transaction also has an associated unique identifier called TID. An *association rule* is an implication of the form $X \Longrightarrow Y$, where $X, Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. X is called the antecedent and Y is called the consequent of the rule.

In general, a set of items (such as the antecedent or the consequent of a rule) is called an *itemset*. The number of items in an itemset is called the *length* of an itemset. Itemsets of some length k are referred to as k–itemsets. For an itemset $X \cdot Y$, if Y is an m–itemset then Y is called an *m-extension* of X.

Each itemset has an associated measure of statistical significance called *support*. For an itemset $X \subset \mathcal{I}$, *support(X)* $= s$, if the fraction of transactions in $\mathcal{D}$ containing X equals s. A rule has a measure of its strength called *confidence* defined as the ratio *support(*$X \cup Y$*) / support(*X*).*

The problem of mining association rules is to generate all rules that have support and confidence greater than or equal to some user specified minimum support and minimum confidence thresholds, respectively. This problem can be decomposed into the following subproblems:

1. All itemsets that have support greater than or equal to the user specified minimum support are generated. These itemset are called the *large* itemsets. All others are said to be *small*.
2. For each large itemset, all the rules that have minimum confidence are generated as follows: for large itemset X and any $Y \subset X$, if *support(*X*)/support(*$X - Y$*)* $\geq$ *minimum_confidence*, then the rule $X - Y \Longrightarrow Y$ is a valid rule.

For example, let $T_1 = \{A, B, C\}$, $T_2 = \{A, B, D\}$, $T_3 = \{A, D, E\}$ and $T_4 = \{A, B, D\}$ be the only transactions in the database. Let the minimum support and minimum confidence be 0.5 and 0.8 respectively. Then the large itemsets are the following: $\{A\}, \{B\}, \{D\}, \{AB\}, \{AD\}, \{BD\}$ and $\{ABD\}$. The valid rules are $B \Longrightarrow A$ and $D \Longrightarrow A$.

The second subproblem, i.e., generating rules given all large itemsets and their supports, is relatively straightforward. However, discovering all large itemsets and their supports is a nontrivial problem if the cardinality of the set of items, $\mid \mathcal{I} \mid$, and the database, $\mathcal{D}$, are large. For example, if $\mid \mathcal{I} \mid = m$, the number of possible distinct itemsets is 2^m. The problem is to identify which of these large number of itemsets has the minimum support for the given set of transactions. For very small values of m, it is possible to setup 2^m counters, one for each distinct itemset, and count the support for every itemset by scanning the database once. However, for many applications m can be more than 1,000. Clearly, this approach is impractical. To reduce the combinatorial search space, all algorithms exploit the following property: any subset of a large itemset must also be large. Conversely, all extensions of a small itemset are also small. This property is used by all existing algorithms for mining association rules as follows: initially support for all itemsets of length 1 (1-itemsets) are tested by scanning the database. The itemsets that are found to be small are discarded. A set of 2-itemsets called *candidate itemsets* are generated by extending the large 1-itemsets generated in the previous pass by one (1-extensions) and their support

is tested by scanning the database. Itemsets that are found to be large are again extended by one and their support is tested. In general, some kth iteration contains the following steps:

1. The set of candidate k-itemsets is generated by 1-extensions of the large $(k-1)$-itemsets generated in the previous iteration.
2. Supports for the candidate k–itemsets are generated by a pass over the database.
3. Itemsets that do not have the minimum support are discarded and the remaining itemsets are called large k-itemsets.

This process is repeated until no more large itemsets are found.

3 Previous Work Using Static Databases

The problem of generating association rules was first introduced in [1] and an algorithm called *AIS* was proposed for mining all association rules. In [4], an algorithm called *SETM* was proposed to solve this problem using relational operations. In [2], two new algorithms called *Apriori* and *AprioriTid* were proposed. These algorithms achieved significant improvements over the previous algorithms. The rule generation process was also extended to include multiple items in the consequent and an efficient algorithm for generating the rules was also presented.

The algorithms vary mainly in (a) how the candidate itemsets are generated; and (b) how the supports for the candidate itemsets are counted. In [1], the candidate itemsets are generated on the fly during the pass over the database. For every transaction, candidate itemsets are generated by extending the large itemsets from the previous pass with the items in the transaction such that the new itemsets are contained in that transaction. In [2] candidate itemsets are generated in a separate step using only the large itemsets from the previous pass. It is performed by joining the set of large itemsets with itself. The resulting candidate set is further pruned to eliminate any itemset whose subset is not contained in the previous large itemsets. This technique produces a much smaller candidate set than the former technique.

Supports for the candidate itemsets are determined as follows. For each transaction, the set of all candidate itemsets that are contained in that transaction are identified. The counts for these itemsets are then incremented by one. Apriori and AprioriTid differ based on the data structures used for generating the supports for candidate itemsets.

In Apriori, the candidate itemsets are compared with the transactions to determine if they are contained in the transaction. A hashtree structure is used to restrict the set of candidate itemsets compared so that subset testing is optimized. Bitmaps are used in place of transactions to make the testing fast. In AprioriTid, after every pass, an encoding of all the large itemsets contained in a transaction is used in place of the transaction. In the next pass, candidate itemsets are tested for inclusion in a transaction by checking whether the large

itemsets used to generate the candidate itemset are contained in the encoding of the transaction. In Apriori, the subset testing is performed for every transaction in each pass. However, in AprioriTid, if a transaction does not contain any large itemsets in the current pass, that transaction is not considered in subsequent passes. Consequently, in later passes, the size of the encoding can be much smaller than the actual database. A hybrid algorithm is also proposed which uses Apriori for initial passes and switches to AprioriTid for later passes.

In [6] we introduced our Partition algorithm, which reduces the number of database scans to only two. In one scan it generates a set of all potentially large itemsets by scanning the database once. This set is a superset of all large itemsets, i.e., it may contain false positives. But no false negatives are reported. During the second scan, counters for each of these itemsets are set up and their actual support is measured in one scan of the database.

The algorithm executes in two phases. In the first phase, the Partition algorithm logically divides the database into a number of non-overlapping partitions. The partitions are considered one at a time and all large itemsets for that partition are generated. At the end of phase I, these large itemsets are merged to generate a set of all potential large itemsets. In phase II, the actual support for these itemsets are generated and the large itemsets are identified. The partition sizes are chosen such that each partition can be accommodated in the main memory so that the partitions are read only once in each phase.

The results reported in [6] show that the number of page reads can be reduced as much as 87% over the Apriori algorithm [2] and that the overall CPU time can be reduced as much as 81%. The details of the specific experiments are in [6].

4 Partition Algorithm for Dynamic Databases

Since our Partition Algorithm divides the database into any number of partitions for processsing, it seems a perfect starting point for developing an algorithm for mining associations in a dynamically changing database.

Definition A *partition* $p \subseteq \mathcal{D}$ of the database refers to any subset of the transactions contained in the database $\mathcal{D}$. Any two different partitions are non-overlapping, i.e., $p_i \cap p_j = \emptyset, i \neq j$. We define *local support* for an itemset as the fraction of transactions containing that itemset in a partition. We define a *local candidate itemset* to be an itemset that is being tested for minimum support within a given partition. A *local large itemset* is an itemset whose local support in a partition is at least the user defined minimum support (e.g., 2 %, 0.005, etc). A local large itemset may or may not be large in the context of the entire database. We define *global support*, *global large itemset*, and *global candidate itemset* as above except they are in the context of the entire database $\mathcal{D}$. Our goal is to find all global large itemsets.

We can envision two partitions, the original database and the set of transactions to be inserted. Since we assume that the mining algorithm was already run on the original database, we need to take advantage of that by modifying the

Partiton algorithm to create a file containing the global large itemsets along with their counts and the number of records in the current database. This information will allow us to reduce the number of scans on the original database to just one. However, we will have to read the file of global large itemsets instead, which will be much smaller than the original database of transactions. For example, with an original database of 100,000 transactions having an average transaction size of 10 items, we used 4,000,000 bytes of storage. For the same set of data, we computed 93 large itemsets with an average size of 4 items. This produces a large itemset file size of 1488 bytes plus the 4 bytes for each of the 93 large itemset counts and 4 bytes for the total number of records, giving a grand total of 1864 bytes. This is a considerable difference in size and subsequent I/O costs for reading.

Besides modifying our Partition algorithm to save the necessary global large itemset information, we must modify the algorithm to take advantage of this information. As mentioned, the Partition algorithm consists of two phases. In the first phase the local large itemsets are computed along with their counts. Since we are treating the original database as partition 1, we do not have to read that for phase I, but instead read the global large itemset file. Not only does this save I/O time but also CPU time since the large itemsets for that partition do not have to be computed. The next step in phase I is to process the second partition, i.e., the new data. A set of added transactions will be treated differently from a set of deleted transactions. Let's consider the insertion case first, which will be more typical. In the case of insertions, the Partition algorithm works exactly the same as the original. It scans partition 2 (i.e., the inserted transactions) and computes the large itemsets and their counts along with the total number of records to be inserted.

For phase II, we need to see if the local large itemsets of partition 1 and partition 2 will be globally large. This requires reading in the original database once and reading in the set of new transactions once (i.e., for the second time). We could take the union of the local large itemsets for partition 1 and partition 2 and do the processing just as the original algorithm does. However, we can reduce the CPU time by only checking the local large itemsets for partition 1 against the set of new transactions and only checking the local large itemsets for partition 2 against the original database. Of course the number of reads is still the same but the number of comparisons is reduced. In addition, the local large itemsets that are contained in the sets from both partition 1 and partition 2 will in fact be globally large and their counts will simply be the sum of the two local counts. So, we can further reduce the number of comparisons needed in phase II. Hence, counts for the local large itemsets, which are not in the intersection of the two partitions' local large itemsets will be computed during a scan of the approriate data. As with the original algorithm, if the global count for the local large itemsets satisfies the support, then we write that large itemset and its count into our new file of global large itemsets. This file will replace our previous global large itemset file.

We already mentioned that a set of deleted transactions will be processed differently from a set of inserted transactions. The problem with deletions is that the file of large itemsets may not contain all the large itemsets after deleting the specified records. It is possible that an itemset in the original database was not large but after deleting a certain number of transactions, it becomes large. For example, consider a database of 100 transactions with a minimum support of 10%. It might be that the original count for a given itemset, I, is 9, which is too low to satisfy the support criteria. Now suppose that 10 transactions are deleted and that those 10 transactions are not the ones used in computing the count for itemset I. With a minimum support of 10%, itemset I, would now be considered a large itemset although it did not appear in the original itemset file. Because of this, the original database, excluding the deleted transactions, must be read and processed in phase I, just as in the original Partition algorithm.

However, we may be able to reduce the CPU time for phase I by using the large itemset file. Consider the following: read in the large itemset file and read in the set of deleted transactions. While reading the set of deleted transactions, use this data to compute the counts, for just those large itemsets stored in the large itemset file. We can then subtract these counts from our original counts for the associated large itemsets. If the counts indicate that the support is no longer satisfied, then we can delete that large itemset. In addition, we can use the information that a former large itemset is no longer large for pruning when we have to process the original database minus the deleted transactions to generate large itemsets. The bottom line is whether reading the large itemset file and doing additional processing with the deleted transactions can actually help prune potentially large itemsets from consideration and reduce the CPU costs at this step to produce an overall savings. We leave this for future study.

Algorithm The Dynamic Partition algorithm is shown in Figure 1. Initially the database $\mathcal{D}$ is logically divided into 2 partitions. Actually the set of transactions to be inserted can be further partitoned but for simplicity of presentation, we consider it as only one partition. Phase I of the algorithm takes 2 iterations. During iteration i only partition p_i is considered. The function `gen_large_itemsets` takes partition p_2 as input and generates local large itemsets of all lengths, $L_1^i, L_2^i, \ldots, L_l^i$ as the output. In phase II, the algorithm sets up counters for each global candidate itemset from L^i and counts their support for the partition $p_{i \bmod 2 + 1}$ and generates the global large itemsets.

The key to the correctness of the above algorithm is the same as our original Partition algorithm which is, any potential large itemset appears as a large itemset in at least one of the partitions. A formal proof is given in [7].

4.1 Generation of Local Large Itemsets

The procedure `gen_large_itemsets` takes a partition and generates all large itemsets (of all lengths) for that partition. The procedure is the same as used in our previous work [6] for the insertion case but differs slightly for the deletion case

```
p1 = the current database of transactions
p2 = the set of transactions to be inserted
L^i = the set of all local large itemsets in partition i
l = an individual candidate large itemset contained in L_i
L^G_old = large itemset file

Phase I
1)  read_in_large_itemset_file(L^G_old) and store in L^1
2)  read_in_partition(p2)
3)  L^2 = gen_large_itemsets(p2)

Phase II
4)  read_in_partition(p1)
5)  for all candidates l ∈ L^2 gen_count(l, p1)
6)  read_in_partition(p2)
7)  for all candidates l ∈ L^1 gen_count(l, p2)
8)  L^G = {l ∈ L^1 ∪ L^2 | l.count ≥ minSup}
9)  write_out_large_itemset_file(L^G_new)
```

Fig. 1. Dynamic Partition Algorithm

with respect to pruning potential large itemsets. The prune step is performed as follows:

```
prune (c: k--itemset)
forall (k − 1)--subsets s of c do
     if s ∉ L_{k−1} then
          return ''c can be pruned''
```

The prune step eliminates extensions of $(k-1)$-itemsets which are not found to be large, from being considered for counting support. For example, if L_3^p is found to be $\{\{1\ 2\ 3\}, \{1\ 2\ 4\}, \{1\ 3\ 4\}, \{1\ 3\ 5\}, \{2\ 3\ 4\}\}$, the candidate generation initially generates the itemsets $\{1\ 2\ 3\ 4\}$ and $\{1\ 3\ 4\ 5\}$. However, itemset $\{1\ 3\ 4\ 5\}$ is pruned since $\{1\ 4\ 5\}$ is not in L_3^p. This technique is the same as the one described in [2] except in our case, as each candidate itemset is generated, its count is determined immediately.

The counts for the candidate itemsets are generated as follows. Associated with every itemset, we define a structure called as *tidlist*. A tidlist for itemset l contains the *TID*s of all transactions that contain the itemset l within a given partition. The *TID*s in a tidlist are kept in sorted order. Clearly, the cardinality of the tidlist of an itemset divided by the total number of transactions in a partition gives the support for that itemset in that partition.

Initially, the tidlists for 1-itemsets are generated directly by reading the partition. The tidlist for a candidate k-itemset is generated by joining the tidlists of the two $(k-1)$-itemsets that were used to generate the candidate k-itemset.

For example, in the above case the tidlist for the candidate itemset {1 2 3 4} is generated by joining the tidlists of itemsets {1 2 3} and {1 2 4}.

Correctness It has been shown in [2] that the candidate generation process correctly produces all potential large candidate itemsets. It is easy to see that the intersection of tidlists gives the correct support for an itemset.

4.2 Generation of Final Large Itemsets

The global candidate set is generated as the union of all local large itemsets from all partitions. In phase II of the algorithm, global large itemsets are determined from the global candidate set. This phase also takes 2 (i.e., the number of partitions) iterations. Initially, a counter is set up for each candidate itemsets and initialized to zero. Next, for each partition, tidlists for all 1-itemsets are generated. The support for a candidate itemset in that partition is generated by intersecting the tidlists of all 1-subsets of that itemset. The cumulative count gives the global support for the itemsets. The procedure `gen_final_counts` is given in Figure 2. Any other technique such as the subset operation described in [2], can also be used to generate global counts in phase II.

```
     C_k^G = set of global candidate itemsets of length k
1)   forall 1-itemsets do
2)      generate the tidlist
3)   for (k = 2; C_k^G ≠ ∅; k++) do begin
4)      forall k--itemset c ∈ C_k^G do begin
5)         templist = c[1].tidlist ∩c[2].tidlist ∩...∩ c[k].tidlist
6)         c.count = c.count + | templist |
7)      end
8)   end
```

Fig. 2. Procedure gen_final_counts

Correctness Since the partitions are non-overlapping, a cumulative count over all partitions gives the support for an itemset in the entire database.

4.3 Discovering Rules

Once the large itemsets and their supports are determined, the rules can be discovered in a straight forward manner as follows: if l is a large itemset, then for every subset a of l, the ratio *support(l) / support (a)* is computed. If the ratio is at least equal to the user specified minimum confidence, them the rule $a \Longrightarrow (l-a)$ is output. A more efficient algorithm is described in [2]. As mentioned earlier, generating rules given the large itemsets and their supports is much simpler compared to generating the large itemsets. Hence we have not attempted to improve this step further.

4.4 Size of the Global Candidate Set

The global candidate set contains many itemsets which may not have global support (false candidates). The fraction of false candidates in the global candidate set should be as small as possible otherwise much effort may be wasted in finding the global supports for those itemsets. The number of false candidates depends on many factors such as the characteristics of the data and the size of the original data and the added data which make up the two partitions in our algorithm.

The local large itemsets are generated for the same minimum support as specified by the user. Hence this is equivalent to generating large itemsets with that minimum support for a database which is the same as the partition. So, for sufficiently large partition sizes, the number of local large itemsets is likely to be comparable to the number of large itemsets generated for the entire database. Additionally, if the data characteristics are uniform across partitions, then a large number of the itemsets generated for individual partitions may be common.

Table 1. Comparison of Global Candidate and Final Large Itemsets (support is 0.5 % and 10% data is added)

Itemset Length	# of Global Candidate Large Itemsets	# of Final Large Itemsets
2	929	323
3	1120	297
4	1197	308
5	1075	301
6	759	224
7	393	122
8	139	45
9	30	10
10	1	1
total	5645	1631

The sizes of the local and global candidate sets may be susceptible to data skew. A gradual change in data characteristics, such the average length of transactions, can lead to the generation of a large number of local large sets which may not have global support. For example, due to severe weather conditions, there may be an abnormally high sales of certain items which may not be bought during the rest of the year. If a new partition is made up of data from only this period, then certain itemsets will have high support for that partition, but will be rejected during phase II due to lack of global support. A large number of such spurious local large itemsets can lead to much wasted effort. Another problem is that fewer itemsets will be found common between partitions leading to a larger global candidate set. These effects can be seen in Table 1 and in Table 2.

Table 2. Comparison of Global Candidate and Final Large Itemsets (support is 0.5 % and 100% data is added)

Itemset Length	# of Global Candidate Large Itemsets	# of Final Large Itemsets
2	361	301
3	452	381
4	538	300
5	398	277
6	228	217
7	122	121
8	45	45
9	10	10
10	1	1
total	2155	1603

In Table 1 we show the variation in the size of the global candidate large itemsets and the final large itemsets produced for differing itemset lengths. The global candidate large itemsets are the result of the union of the local large itemsets from the original data and the added data. The original database contained 13,000 transactions and the added data contained only 1300 transactions. The minimum support was set at 0.05 %. We can compare the results in Table 1 with those shown in Table 2. It can be seen from the two tables that as the size of the added data becomes larger, the variation in the sizes of the global candidate and final large itemsets is reduced dramatically. It should be noted that when the partition sizes of the original and added data are sufficiently large, the local large itemsets and the global candidate itemsets are likely to be very close to the actual large itemsets as it tends to eliminate the effects of local variations in data.

5 Performance Comparison

In this section we describe the experiments and the performance results of our dynamic algorithm and our standard approach. The experiments were run on a Sun Sparc workstation. All the experiments were run on synthetic data. For the experiments, we used the same method for generating the synthetic data sets as in [6].

5.1 Synthetic Data

The synthetic data is said to simulate a customer buying pattern in a retail environment. The length of a transaction is determined by poisson distribution with mean μ equal to $|T|$. The transaction is repeatedly assigned items from a set of potentially maximal large itemsets, $\mathcal{T}$ until the length of the transaction

does not exceed the generated length. The average size of a transaction is 10 items and the average size of a maximal potentially large itemset is 4 items.

The length of an itemset in $\mathcal{T}$ is determined according to poisson distribution with mean μ equal to $|I|$. The items in an itemset are chosen such that a fraction of the items are common to the previous itemset determined by an exponentially distributed random variable with mean equal to a correlation level. The remaining items are randomly picked. Each itemset in $\mathcal{T}$ has an exponentially distributed weight that determines the probability that this itemset will be picked. Not all items from the itemset picked are assigned to the transaction. Items from the itemset are dropped as long as an uniformly generated random number between 0 and 1 is less than a corruption level, c. The corruption level for itemset is determined by a normal distribution with mean 0.5 and variance 0.1.

5.2 Experimental Results

Two different data sets were used for performance comparison. One was a 2 MB file consisting of 50,000 transactions and the other a 1 MB consisting of 25,000 transactions. For both data sets the number of items was set to 1,000.

Figure 3 shows the execution times for the two synthetic datasets. The execution times increase for both the Incremental and Standard algorithms as the minimum support is reduced because the total number of large and candidate itemsets increases. In general, the execution time increases for both the Incremental and Standard algorithms as the minimum support is reduced. The main reason for this is the fact that the total number of candidate and large itemsets increases. In addition, we have already seen that a relatively small size of added data can produce a large number of locally large itemsets during phase I of the algorithm. This increases the cost during phase II when those itemsets are verified against the original database. So, even though the number of final large itemsets is not increased, there is additional time required for testing the false candidate itemsets. As mentioned, this depends heavily on the characteristics of the data.

From the experimental results, we see that a support of 2% and a support of 1% yield approximately the same number of local large candidate itemsets, producing similar execution times, as shown in Figure 4. In the case of 0.5% support, a noticable increase in the number of candidate itemsets and the number of final itemsets was observed, producing a larger execution time. In Table 3 we show the average improvement of the Incremental approach over the Standard approach.

In Figure 5, we show the local large itemsets generated for both the original database and the added data. Since the database is much larger than the added data and closer in size to the final database size after insertions are done, the number of local large itemsets is much closer to the final number of local large itemsets. We see that the number of local large itemsets peaks at 4, which is our average local large itemset size.

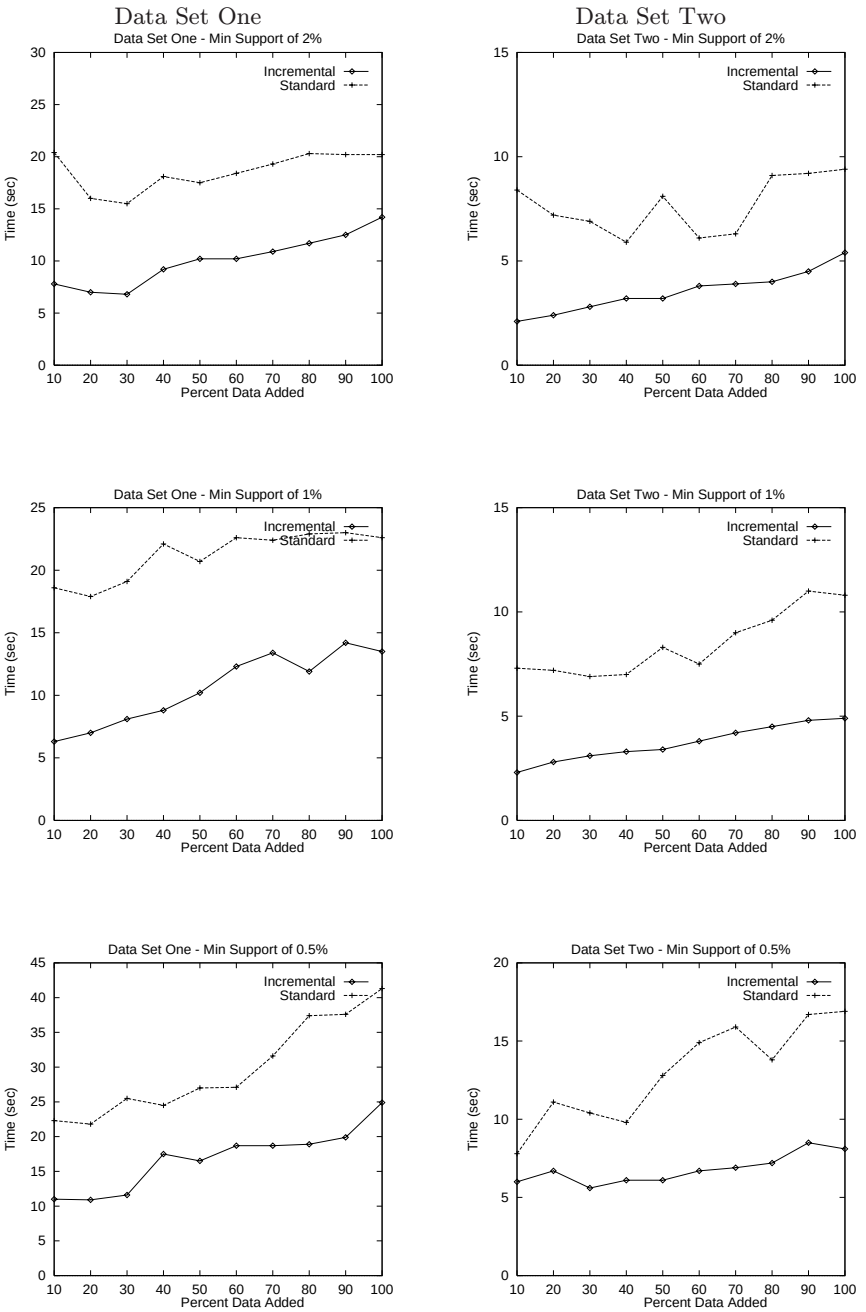


Fig. 3. Execution times

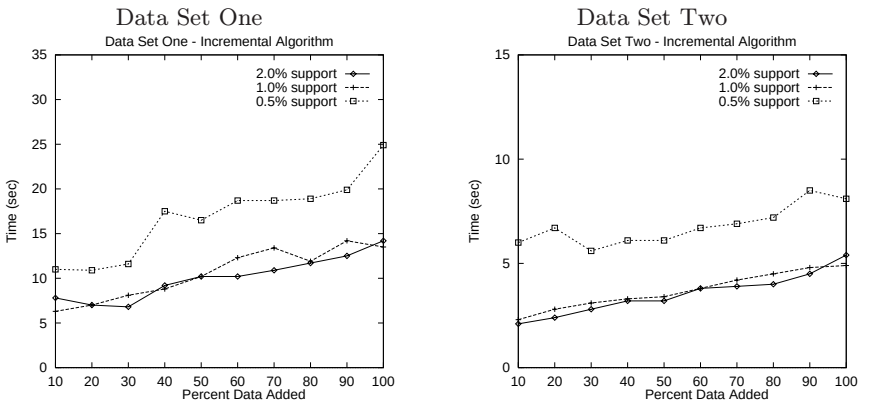


Fig. 4. Execution times for Incremental Algorithm

Table 3. Average Improvement in Execution Time of Incremental Algorithm over Standard Algorithm

Dataset	Minimum Support (%)	Avg. Improvement (%)
1	2.0	46.1
1	1.0	50.7
1	0.5	42.7
2	2.0	52.6
2	1.0	55.0
2	0.5	45.9

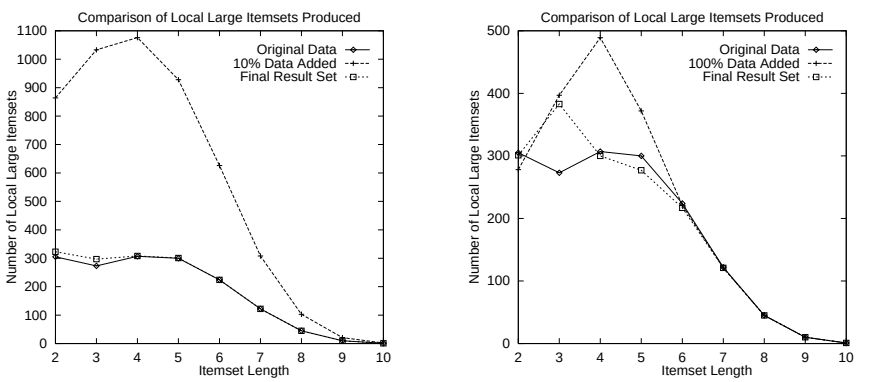


Fig. 5. Large Itemset Comparison

6 Conclusions

We have described an algorithm for dynamic data mining, which is an extension to our previous work for mining associations in static databases. We have run several experiments comparing the execution time of our Incremental algorithm and the Standard algorithm. The average improvement (or reduction in execution time) by using our Incremental algorithm ranged from 42.7% to 55%. In our experiments, the database was set to a size such that the Standard algorithm required only one pass over the data. This is the best we can achieve in reducing I/O costs. Where as, our Incremental algorithm required, in phase I, one pass over the added data and one pass over the stored large itemset file, and in phase II, our algorithm required one pass over the original database and one pass over the added data. As we have observed, the major savings from our algorithm comes about by the fact that the large itemsets for the original database do not have to be re-computed. The substantial savings in execution time makes our Incremental algorithm a desired approach for dynamic data mining.

References

1. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, DC, May 26-28 1993. 49, 50, 50, 52, 52
2. R. Agrawal and R. Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, Santiago, Chile, August 29-September 1 1994. 50, 50, 50, 52, 52, 53, 56, 57, 57, 57
3. J. Han and Y. Fu. Discovery of multiple-level association rules from large databases. In *Proceedings of the VLDB Conference*, pages 420 – 431, September 1995. 50
4. M. Houtsma and A. Swami. Set-oriented mining of association rules. In *Proceedings of the International Conference on Data Engineering*, Taipei, Taiwan, March 1995. 50, 52
5. J. S. Park, M-S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pages 229 – 248, San Jose, California, May 1995. 50
6. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 688–192, Zurich, Swizerland, August 1995. 50, 50, 53, 53, 53, 55, 59
7. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. Technical Report GIT-CC-95-04, Georgia Institute of Technology, Atlanta, GA 30332, January 1995. 55

Efficient Nearest-Neighbour Searches Using Weighted Euclidean Metrics

Ruth Kurniawati, Jesse S. Jin, and John A. Shepherd

School of Computer Science and Engineering
University of New South Wales
Sydney 2052, Australia ruthk@cse.unsw.edu.au
Telephone: 61-2-9385-3989; Fax: 61-2-9385-1813

Abstract. Building an index tree is a common approach to speed up the k nearest neighbour search in large databases of many-dimensional records. Many applications require varying distance metrics by putting a weight on different dimensions. The main problem with k nearest neighbour searches using weighted euclidean metrics in a high dimensional space is whether the searches can be done efficiently. We present a solution to this problem which uses the bounding rectangle of the nearest-neighbour disk instead of using the disk directly. The algorithm is able to perform nearest-neighbour searches using distance metrics different from the metric used to build the search tree without having to rebuild the tree. It is efficient for weighted euclidean distance and extensible to higher dimensions.

Keywords. nearest-neighbour search, weighted Euclidean distance, coordinate transformations, R-trees, spatial access methods

1 Introduction

The problem of k nearest-neighbour search using a distance metric can be stated as follows: given a collection of vectors and a query vector find k vectors closest to the query vector in the given metric. Nearest-neighbour search is an important operation for many applications. In image databases utilizing feature vectors [13,12,7], we want to find k images similar to a given sample image. In instance-based learning techniques [1,14], we would like to find a collection of k instance vectors most similar to a given instance vector in terms of target attributes. Other applications include time-series databases [6], non-parametric density estimation [9], image segmentation [2], etc.

In the k-nearest-neighbour search process, we have a region defined by the k-th farthest neighbour found so far. Depending on the distance metric used, this area could be a circle (for unweighted euclidean distance), diamond (for manhattan distance), ellipse (for weighted euclidean distance), or other shapes. At the beginning of the search process, this area is initialized to cover the whole data space. As we find closer neighbours, the area will get smaller. We call this

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 64–76, 1998.

area "the nearest-neighbour disk"[1]. The search proceeds downward from the top of the tree, checking tree nodes covering smaller area as it deepens. We usually use a branch-and-bound algorithm and try to choose the nodes which are most likely to contain the nearest neighbour first. This is done in order to shrink the nearest-neighbour disk as soon as possible. We can safely ignore nodes whose bounds do not intersect the nearest-neighbour disk.

Many algorithms have been proposed for finding k nearest neighbours more efficiently than a sequential scan through data. These algorithms involve building a spatial access tree, such as an R-tree [10], PMR quadtree [11], k-d-tree [3], SS-tree [22], or their variants [18,8,19,16]. Spatial access methods using such trees generally assume that the distance metric used for building the structures is also the one used for future queries. However, many applications require varying distance metrics. For example, when the the similarity measure is subjective, the distance metric can vary between different users. Another example comes from the instance-based learning [1,14], where the distance metric is learnt by the algorithm. The distance metric will change as more instances are collected. The tree structures above cannot be easily used if the distance metric in the query is different to the distance metric in the index tree. One approach is to map the index tree into the new metric space, but this is equivalent to rebuilding the tree, and therefore, computationally expensive. Another approach is to map the k nearest neighbour bounding disk into the index metric. For example, if we do a query using the Minkowski distance metric on a tree built in euclidean distance, the corresponding nearest neighbour bounding region will be of a diamond shape. Similarly, a nearest neighbour disk in the euclidean distance with different weighting will produce an ellipse. Because of the wide variety of shapes than can be produced, calculating the intersection of the new nearest neighbour bounding envelope and the bounding envelopes of index tree nodes is not trivial and can be computationally expensive.

In this paper, we present an efficient k-nearest-neighbour search algorithm using weighted euclidean distance on an existing spatial access tree. We start by describing a commonly used approach [4]. We identify the drawbacks of this approach and introduce our method. The theoretical foundations and detailed implementation of our method in two dimensional space are given. We also give an extension of the algorithm in higher dimensional spaces. The experiments (up to 32 dimensions) were carried out on an R-tree variant, the SS$^+$-tree [16], (although the proposed approach can be used in any hierarchical structure with a bounding envelope for each node).

[1] We use the 2-dimensional term "disk" regardless of the number of dimensions. To be strict, we should use disk only for 2-dimensional spaces, ellipsoid for 3-dimensional spaces and hyper-ellipsoid for spaces with dimensionality greater than 3.

2 Methods for finding the k nearest-neighbours in weighted euclidean distance.

Tree structured spatial access methods organize the data space in a hierarchical fashion. Each tree node in these structures has a bounding hyper-box (R-trees), bounding hyper-planes (k-d-trees), or a bounding hyper-sphere (SS-trees), giving the extent of its child nodes. The root node covers the whole data space which is split among its children. The process is repeated recursively until the set of points in the covered space can fit in one node. To simplify the discussion, we use two dimensional terms for all geometric constructs in the rest of this section and next section.

2.1 Faloutsos' algorithm for k-nn search in a weighted euclidean distance

If the distance metric in the query is consistent with the metric in the index tree, detecting the intersection between the k nearest-neighbour disk and a node can be performed by a simple comparison of the distance of the k-th neighbour to the query vector and the range of the node's bounds in each dimension. When the distance metric is not consistent with the metric in the tree structure, a commonly used technique [4,5,15] to find the k-nearest-neighbours is as follows (For easy understanding, we take an example, in which the index tree is built in unweighted euclidean distance and the query is issued in a weighted euclidean distance, and visualize the process in Fig. 1):

- First, we define a distance metric which is a lower bound to the weighted distance. It can be a simple transform to both metrics in the query and in the index tree. Usually, the same metric as in the index tree is used.
- We then search for k nearest neighbours to obtain a bound (LBD in Fig. 1).
- Transferring the distance metric to the query distance metric, we obtain a new bounding shape for the k-nn disk (ED in Fig. 1). The maximum distance D can then be calculated.
- Next, we issue a range query to find all vectors within the distance D from the query point (All vectors in RQD, as shown in Fig. 1). The number of vectors returned from the range query will be greater or equal to k.
- We sort the set from the last step and find the true k nearest neighbours.

From Fig. 1, we can see that the circular range query area (RQD) is much larger than the actual k nearest-neighbour disk. This is not desirable, since a larger query area means we have to examine more tree nodes.

Another drawback of the approach is that we have to search the tree twice. The first scan finds the k nearest-neighbours and the second scan finds all the vectors located within the range D (a range query). Sorting is also needed for selecting first k nearest neighbours within range D.

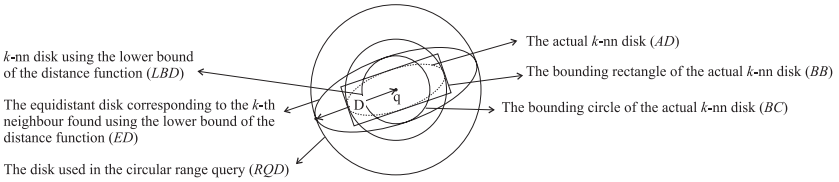


Fig. 1. The evolution of search areas in Faloutsos' approach and our bounding rectangle.

2.2 An efficient *k*-nn search in variable metrics using the bounding rectangle

We propose an efficient k-nn search with variable metrics. The approach is based on the observation that a transform between the query metric and the index metric exists and transforming the k-nn disk is much simpler than transforming all tree nodes. Using this transformation, we can find the length of the major axis of the k nearest-neighbour disk[2]. However, conducting the search using the ellipse directly is not always desirable. For d-dimensional vectors we will have to calculate the intersection of $(d-1)$-dimensional hyper-planes and a d-dimensional hyper-ellipse. If the intersection is not empty, the result will be a $(d-1)$-dimensional hyper-ellipse. Hence in our solution we use a simpler shape to "envelope" the k nearest-neighbour disk.

We propose a rectangular envelope (the bounding rectangle BB of the k nearest-neighbour disk is shown in Fig. 1). The bounding rectangle can be obtained from calculating the length of all axes of the k-nearest-neighbour disk. Another simpler alternative is by calculating the bounding circle of the k nearest-neighbour disk (BC in Fig. 1). The radius of the bounding circle is equal to the half of ellipse's major axis. Theoretical analysis and detailed calculation will be described in Sect. 3.

Comparing with Faloutsos' algorithm, our approach has two advantages. The bounding rectangle is much compact than Faloutsos' query disk (see RQD and BB/BC in Fig. 1). The advantage is significant if the ratio between the largest and the smallest axes of the nearest neighbour disk is large. There is no need for the second search which is also a significant saving if the level of the tree is high.

3 Theory and implementation

In this paper, we will limit our discussion to the *weighted euclidean distance* $\mathrm{d}(\mathbf{x},\mathbf{y},\mathbf{W}) = \sqrt{((\mathbf{x}-\mathbf{y})^T\mathbf{W}(\mathbf{x}-\mathbf{y}))}$ with a symmetrical weight matrix $\mathbf{W}$. The approach described in this paper can be adapted directly to distances in the

[2] Some books define the axis length of an ellipse as the distance from the centre to perimeter along the axis. The definition we use is the distance from perimeter to perimeter along that axis.

class $\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W}) = \sum_{i=1}^{d}(x_i - y_i)^p$, where d is the number of the dimensions and $p = 1, 2, 3, \ldots$. We require that the matrix $\mathbf{W}$ is positive definite, i.e. it has positive pivots, which implies that we will always be able to decompose the matrix $\mathbf{W}$ into $\mathbf{Q}^T\mathbf{Q}$ using Cholesky factorization [20]. In practice, we usually deal with symmetrical weight matrices, since non-symmetrical weight matrices means that the distance measured from $\mathbf{x}$ to $\mathbf{y}$ is different from that measured from $\mathbf{y}$ to $\mathbf{x}$.

As have been defined informally in Sect. 2, the *nearest-neighbour disk* is the area bounded by the equidistant points with the current k-th nearest point to the query measured using a particular distance metric. The term "point" used here and in the rest of the paper is interchangeable with "vector".

A particular distance metric will define a *space* where the distance is used for measurements. An euclidean distance defines an euclidean space. The k nearest-neighbour disk is circular in the same euclidean space. If the Euclidean metric in query has a different weight on the metric used in the index, the k nearest-neighbour disk is an ellipse in that space.

3.1 Theoretical foundations

Assume we have two points $\mathbf{x_1}$ and $\mathbf{y_1}$ in a space S_1 where we use unweighted euclidean distance (see Fig. 2). Applying a transformation defined by the matrix $\mathbf{Q}$ will bring every point in S_1 into another space S_2. Suppose $\mathbf{x_1}$ and $\mathbf{y_1}$ are mapped into $\mathbf{x_2}$ and $\mathbf{y_2}$, respectively. If the transformation includes scaling and rotation operations, then the distance between $\mathbf{x_1}$ and $\mathbf{y_1}$ in S_1 will equal a weighted euclidean distance between $\mathbf{x_2}$ and $\mathbf{y_2}$ where the weight matrix $\mathbf{W}$ equals $\mathbf{Q}^T\mathbf{Q}$. This means that a circle in S_1 will be transformed into an ellipse in S_2 since the equivalent distance in S_2 is a weighted euclidean distance.

Observation 1. *The weighted distance* $d(\mathbf{x_2}, \mathbf{y_2}, \mathbf{W})$ *with* $\mathbf{W} = \mathbf{Q}^T\mathbf{Q}$ *equals* $d(\mathbf{x_1}, \mathbf{y_1}, \mathbf{I})$*, where* $\mathbf{x_2} = \mathbf{Q}\mathbf{x_1}$*,* $\mathbf{y_2} = \mathbf{Q}\mathbf{y_1}$*, and* $\mathbf{I}$ *is the identity matrix.*

Proof. Let $\mathbf{W} = \mathbf{Q}^T\mathbf{Q}$, $\mathbf{x_1} = \mathbf{Q}\mathbf{x_2}$, and $\mathbf{y_1} = \mathbf{Q}\mathbf{y_2}$.

$$\begin{aligned}
\mathrm{d}(\mathbf{x_2}, \mathbf{y_2}, \mathbf{W}) &= \sqrt{(\mathbf{x_2} - \mathbf{y_2})^T\mathbf{W}(\mathbf{x_2} - \mathbf{y_2})} \\
&= \sqrt{(\mathbf{x_2} - \mathbf{y_2})^T\mathbf{Q}^T\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2})} = \sqrt{(\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2}))^T\mathbf{Q}(\mathbf{x_2} - \mathbf{y_2})} \\
&= \sqrt{(\mathbf{Q}\mathbf{x_2} - \mathbf{Q}\mathbf{y_2})^T(\mathbf{Q}\mathbf{x_2} - \mathbf{Q}\mathbf{y_2})} = \sqrt{(\mathbf{x_1} - \mathbf{y_1})^T(\mathbf{x_1} - \mathbf{y_1})} \\
&= \mathrm{d}(\mathbf{x_1}, \mathbf{y_1}, \mathbf{I})
\end{aligned}$$

The transformation matrix $\mathbf{Q}$ defines the scaling and rotation operations which map the circular k nearest-neighbour disk in unweighted euclidean space into an ellipse in the weighted euclidean space. This ellipse is defined by all axes and the query point. As described in Sect. 2, to obtain the bounding box, we need to calculate the length of all the axes and the orientation of the major axis. The solution can be derived from Theorem 2.

Theorem 2. *If we apply a transformation* $\mathbf{Q}$ *to all points of a circle (with radius* r*), the resulting points form an ellipse whose centre is the same as the circle and the length of its axes equals* $2r$ *times the square root of eigenvalues of* $\mathbf{Q}^T\mathbf{Q}$*.*

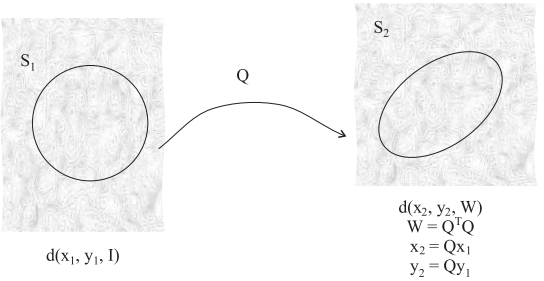


Fig. 2. The transformation between a normal euclidean space and a weighted euclidean space.

Proof. We provide the proof for the two dimensional case. A general proof can be found in [17].

The proof of this theorem depend on a lemma stating that *a real symmetric $n \times n$ matrix has n real eigenvalues, and n linearly independent and orthogonal eigenvectors.* The proof of the lemma can be found in [21].

Let $\mathbf{k_2} = \mathbf{Qk_1}$ where $\mathbf{k_1}$ is an arbitrary point in our original space (S_1 as shown in Fig. 2). Let the circle centre $\mathbf{q_2} = \mathbf{Qq_1}$. The squared distance of $\mathbf{k_2}$ and $\mathbf{q_2}$ is $(\mathbf{Qk_1} - \mathbf{Qq_1})^T \cdot (\mathbf{Qk_1} - \mathbf{Qq_1}) = (\mathbf{k_1} - \mathbf{q_1})^T \mathbf{Q}^T \mathbf{Q}(\mathbf{k_1} - \mathbf{q_1})$. Since $\mathbf{Q}^T\mathbf{Q}$ is real and symmetric, according to the lemma, it has two real eigenvalues and two orthogonal eigenvectors. Let $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ be the unit eigenvectors corresponding to eigenvalues λ_1 and λ_2, respectively.

These orthogonal eigenvectors span $\Re^2$, and hence $(\mathbf{k_1} - \mathbf{q_1})$ can be written as a linear combination of $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$. Suppose $(\mathbf{k_1} - \mathbf{q_1}) = r\cos(\theta)\boldsymbol{e}_1 + r\sin(\theta)\boldsymbol{e}_2$.

$$
\begin{aligned}
&(\mathbf{k_2} - \mathbf{q_2})^T(\mathbf{k_2} - \mathbf{q_2}) = (\mathbf{k_1} - \mathbf{q_1})^T \mathbf{Q}^T \mathbf{Q}(\mathbf{k_1} - \mathbf{q_1}) \\
&= (r\cos\theta \boldsymbol{e}_1^T + r\sin\theta \boldsymbol{e}_2^T)(r\cos\theta \mathbf{Q}^T\mathbf{Q}\boldsymbol{e}_1 + r\sin\theta \mathbf{Q}^T\mathbf{Q}\boldsymbol{e}_2) \\
&= (r\cos\theta \boldsymbol{e}_1^T + r\sin\theta \boldsymbol{e}_2^T)(r\cos\theta \lambda_1\boldsymbol{e}_1 + r\sin\theta \lambda_2\boldsymbol{e}_2) \\
&= r^2\lambda_1\cos^2\theta + r^2\lambda_2\sin^2\theta
\end{aligned}
$$

Square distance $r^2\lambda_1\cos^2\theta + r^2\lambda_2\sin^2\theta$ has its extreme values at $\theta =$ multiplies of $\frac{\pi}{2}$ – that is, when we are in $\boldsymbol{e}_1$ or $\boldsymbol{e}_2$ directions. The values at those points are $r^2\lambda_1$ and $r^2\lambda_2$, respectively. Using our definition of the ellipse axis, the length of two axes are $2r\sqrt{\lambda_1}$ and $2r\sqrt{\lambda_2}$, respectively.

□

We can write the weight matrix in diagonal form: $\mathbf{W} = \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{-1}$, where $\mathbf{E}$ is an orthonormal matrix whose columns are the eigenvectors of $\mathbf{W}$ and $\boldsymbol{\Lambda}$ is a diagonal matrix of the corresponding eigenvalues. Since $\mathbf{E}$ is orthonormal, then $\mathbf{E}^{-1} = \mathbf{E}^T$. Hence:

$$
\begin{aligned}
\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W}) &= \sqrt{((\mathbf{x} - \mathbf{y})^T \mathbf{W}(\mathbf{x} - \mathbf{y}))} \\
&= \sqrt{((\mathbf{x} - \mathbf{y})^T \mathbf{E}\boldsymbol{\Lambda}\mathbf{E}^{-1}(\mathbf{x} - \mathbf{y}))} \\
&= \sqrt{((\mathbf{E}^{-1}\mathbf{x} - \mathbf{E}^{-1}\mathbf{y})^T \boldsymbol{\Lambda}(\mathbf{E}^{-1}\mathbf{x} - \mathbf{E}^{-1}\mathbf{y}))}
\end{aligned}
$$

We can see the last line of the equation as another weighted distance with diagonal weight matrix $\mathbf{\Lambda}$. Because $\mathbf{\Lambda}$ is diagonal, the nearest neighbour disk in this space has axes parallel to the coordinate axes. The size of the nearest neighbour disk is the same in both spaces since $\mathbf{E}$ is orthonormal. Suppose $\mathbf{x}_\mathrm{l}$ and $\mathbf{y}_\mathrm{l}$ are in this space, then $\mathbf{x}_\mathrm{l} = \mathbf{E}^{-1}\mathbf{x}$ and $\mathbf{y}_\mathrm{l} = \mathbf{E}^{-1}\mathbf{y}$. We can see $\mathbf{E}$ as the rotation matrix that rotates the nearest neighbour disk. Since $\mathbf{x} = \mathbf{E}\mathbf{x}_\mathrm{l}$ and $\mathbf{y} = \mathbf{E}\mathbf{y}_\mathrm{l}$, then the desired rotation is $\mathbf{E}$. The length of the disk's axis in the direction given by the i-th column of $\mathbf{E}^{-1}$ is given by $\sqrt{(\mathrm{d}(\mathbf{x}, \mathbf{y}, \mathbf{W})/\lambda_i)}$.

3.2 Calculating the bounding circle of the k nearest-neighbour disk

In a k nearest-neighbour search using weighted euclidean distance, we start from an initial disk covering the whole data space. After we find k neighbours from searched nodes, a new (smaller) k-nn disk is obtained. We have to calculate the bounding circle of the k-nn disk every time we find a neighbour nearer to the query point than the k-th nearest neighbour found so far. The problem can be stated as follows: given the query point $\mathbf{q}$ and the newly found k-th neighbour we need to calculate the radius of the bounding circle of the ellipse with $\mathbf{q}$ as its centre and $\mathbf{k}$ on its perimeter (see Fig. 3).

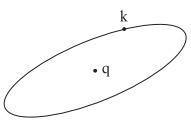


Fig. 3. The query point $\mathbf{q}$ and its k-th nearest-neighbour $\mathbf{k}$.

Based on the discussion in Sect. 3.1, the calculation of the bounding circle of the k nearest-neighbour disk is done as follows.

- Let λ be the largest eigenvalue of matrix $\mathbf{W}$.
- The bounding circle is a circle with the query point $\mathbf{q}$ as its centre and r as its radius, where $r = \mathrm{d}(\mathbf{q}, \mathbf{k}, \mathbf{W})/\sqrt{\lambda_{\min}}$, where $\lambda_{\min}$ is the smallest eigenvalue.

3.3 Calculating the bounding rectangle of the k nearest-neighbour disk

Instead of using bounding circle, we also can use bounding rectangle. The problem, similar to the one described in Sect. 3.2 can be stated as follows: given a query point $\mathbf{q}$ and the newly found k-th neighbour $\mathbf{k}$, calculate the length, the sides and the orientation of the bounding rectangle of the ellipse centred at $\mathbf{q}$ with $\mathbf{k}$ on its perimeter (see Fig. 3).

Let $\mathbf{W}$ is the weight matrix, and $\mathbf{I}$ is the identity matrix, $\mathbf{q}$ is the query point, $\mathbf{k}$ is the k-th nearest neighbour found, $\mathbf{E}$ is an orthonormal matrix whose columns is the eigenvectors of $\mathbf{W}$, λ_i is the eigenvalues corresponding to the

eigenvector in column i of $\mathbf{E}$, and $\boldsymbol{e}_i$ is the vector in column i of $\mathbf{E}^{-1}$. The bounding rectangle of the nearest-neighbour disk can be calculated as follows.

- Let $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ be the eigen-vectors of $\mathbf{W}$ and let λ_1 and λ_2 be the corresponding eigen-values sorted according to their magnitudes.
- Then the bounding rectangle R is defined by the centre $\mathbf{q}$ and dimensional vectors $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ where $r = \mathrm{d}(\mathbf{q}, \mathbf{k}, \mathbf{W})$, and $\boldsymbol{v}_1 = \boldsymbol{e}_1 r/\sqrt{\lambda_1}$ and $\boldsymbol{v}_2 = \boldsymbol{e}_2 r/\sqrt{\lambda_2}$. The corners of the rectangles are:

$$\begin{aligned}\mathbf{r}_1 &= \mathbf{q} + \boldsymbol{v}_1 + \boldsymbol{v}_2\\ \mathbf{r}_2 &= \mathbf{q} + \boldsymbol{v}_1 - \boldsymbol{v}_2\\ \mathbf{r}_3 &= \mathbf{q} - \boldsymbol{v}_1 - \boldsymbol{v}_2\\ \mathbf{r}_4 &= \mathbf{q} - \boldsymbol{v}_1 + \boldsymbol{v}_2\end{aligned}$$

Vectors $\boldsymbol{v}_i$ are in the directions parallel to the axes of the nearest neighbour disk. The fact that $\boldsymbol{v}_i$ is equal to $\boldsymbol{e}_i r/\sqrt{\lambda_i}$ can is explained in Sect. 3.1. Note that the rectangle R can be defined by the centre point $\mathbf{q}$ and the directional vectors denoted as $R(\mathbf{q}, \boldsymbol{v}_1, \boldsymbol{v}_2)$ and does not necessarily have any side parallel to the coordinate axes.

In high dimensional space, care must be taken in calculating the intersection between the bounding hyper-box and tree's bounding envelopes. A naïve calculation is exponential to the dimension. We provide an algorithm with time complexity $\mathrm{O}(d^3)$ of dimension d in Sect. 4.

4 Extension to a higher dimensional space

The algorithm in Sect. 3.3 can be used directly in space with dimensionality greater than 2. It contains three major computations, calculating Cholesky decomposition, finding the bounding hyper-box of the new k-nn hyper-ellipsoid, and checking the intersection of a node and the hyper-box. In this section we describe an efficient extension which gives $\mathrm{O}(d^3)$, $\mathrm{O}(d^2)$ and $\mathrm{O}(d^3)$ complexities to three steps respectively.

Note that checking if a node is intersected with the bounding hyper-box does not need to calculate the corner points of the bounding hyper-box. Otherwise, the approach will not be able to scale up to high dimensions because the number of corners of a hyper-box in a d-dimensional space is 2^d. We can define a bounding hyper-box using a centre point and d directional vectors $(\boldsymbol{r}_i)$. Since the Cholesky decomposition and the eigen matrix are the same throughout the search process, we only need to keep the radius r. The centre of the hyper-box will be the query point $\mathbf{q}$ itself. The cost to calculate the eigen matrix and Cholesky decomposition of a d x d matrix is $\mathrm{O}(d^3)$. We only need to recalculate these matrices whenever the weight matrix changes, which can only happen between queries. The calculation of the bounding rectangle for the query area can be done in $\mathrm{O}(d^2)$.

Our algorithm to check the intersection of tree nodes and the bounding hyper-box of the nearest-neighbour disk has a complexity $\mathrm{O}(d^3)$. If the bounding

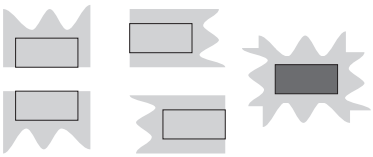


Fig. 4. A bounding rectangle is the intersection of four half-spaces.

hyper-box has sides parallel/perpendicular to the coordinate axes, intersections can be calculated in O(d). We only need to calculate the intersection of each dimension. It can be done by utilizing the fact that a d-dimensional hyper-box is actually an intersection of $2d$ hyper-planes (as shown in the 2D case in Fig. 4).

Suppose the query's bounding hyper-box B is defined by its centre point $\mathbf{q}$ and d orthogonal vectors $\boldsymbol{v}_i$, ie, $B(\mathbf{q}, \boldsymbol{v}_1, ..., \boldsymbol{v}_d)$. The bounding hyper-box of a tree node is A and its closest corner to the origin is $\mathbf{a_{min}}$ and the farthest corner to the origin is $\mathbf{a_{max}}$. We use $\mathbf{r}(j)$ and $\boldsymbol{r}(j)$ to represent the projections of point $\mathbf{r}$ and vector $\boldsymbol{r}$ in dimension j respectively in the following description.

- For all dimensions $j = 1..d$, find the minimum coordinate of the query's bounding hyper-box (can be done in O(d) by subtracting $\boldsymbol{v}_i(j)$ from $\mathbf{q}(j)$). If any minimum is not inside the half-space defined by the equation $x \leq \mathbf{a_{max}}(j)$, then we do not have an intersection. Otherwise, continue checking. This process takes O(d^2) comparisons.
- For all dimension $j = 1..d$, find the maximum coordinate of the query's bounding hyper-box (can be done in O(d) by adding $\boldsymbol{v}_i(j)$ to $\mathbf{q}(j)$). If any minimum is not inside the half-space defined by the equation $x \geq \mathbf{a_{min}}(j)$, then we do not have an intersection. Otherwise, continue checking. This process also takes O(d^2) comparisons.
- If above checking fails, it does not mean that we have a non-empty intersection area. It means that B is not intersected with the hyper-box $\mathbf{a_{min}} \leq x \leq \mathbf{a_{max}}$. The check should also be made from the B rectangle in the query space using all the half-spaces defined by $x \leq \mathbf{q}+\boldsymbol{v}_i$ and $x \geq \mathbf{q}-\boldsymbol{v}_i$. Transformation to the query space takes O(d^3) multiplication and rest comparisons are O(d^2). The total complexity is O(d^3) + O(d^2) = O(d^3).

5 Empirical tests and discussion

We implemented the algorithm described in Sect. 3.3and 3.2 on top of an R-tree variant, the SS$^+$-tree [16]. We compare our algorithms with Faloutsos' method using the unweighted euclidean distance as the lower bound of the weighted distance. We evaluate the performance of the methods using the number of nodes touched/used. For all the experiments in this paper we choose to expand the node whose centroid is closest to the query point with respect to the currently used distance metric. Based on our experience [16], this criterion truncates branches

more efficiently than other criteria, e.g. the minimum distance and the minimum of the maximum distance [18].

All the experiments in Table 1 are in 30 dimensions using 14,016 texture feature vectors extracted from images in Photo Disc's image library using the Gabor filter [12]. The experiments was done using a weight matrices whose ratio between the largest and smallest eigenvalues range from 2 to 32. The fan out of tree's leaf nodes is 66 and the fan out of tree's internal nodes is 22.

We measured the number of leaf nodes that has to be examined (*ltouched*), the number of internal nodes accessed (*inode*), and the number of leaf nodes that actually contain any of the nearest-neighbour set (*lused*). The ratio column is the ratio between square root of the largest and the smallest eigenvalues. All the experiments are 21-nearest-neighbour searches and the tabulated results are the average of 100 trials using a random vector chosen from the dataset.

Table 1. A comparison between the transformation and lower-bound method for the texture vectors of Photo Disc's images (14016 vectors, 30 dimensions, node size: 8192 bytes).

	Bounding (hyper)box			Bounding (hyper)sphere			Faloutsos' method		
ratio	ltouch	lused	inode	ltouch	lused	inode	ltouch	lused	inode
2	25.32	7.06	4.22	19.08	7.32	4.20	36.62	13.78	8.16
4	23.38	6.43	4.17	17.36	7.02	4.24	32.54	12.67	8.09
8	26.04	6.45	4.45	18.04	7.00	4.11	35.63	12.66	8.51
16	24.53	6.76	4.08	17.76	7.16	4.24	33.92	13.43	7.97
32	22.89	6.49	4.15	18.48	7.15	4.12	30.08	12.59	8.01

As can be seen in Table 1, our method outperformed Faloutsos' method in all the ratio values we tried. The number of leaf nodes used (*lused*) by our method is 50% of the number used by Faloutsos' method because our method only need to check the tree once. The same explanation applies to the number of internal nodes accessed (*inode*). The number of leaf nodes that have to be accessed by our method is consistently below 75% of the total leaf nodes accessed by the other method. The data suggest that the ratio between the largest and the smallest eigenvectors does not have any effect in the number of nodes accessed for the texture vectors.

The result experimental results using bounding spheres are somewhat counter intuitive. From our intuition, the results should be worse than the ones using bounding boxes. In two and three dimensions, the bounding rectangle/box of an disk occupy a smaller area/volume than the disk's bounding circle/sphere. Our analytical results [17] supports this empirical results. It shows that in high-dimensional space the query sensitive area resulting from using bounding hyper-spheres is smaller than the one resulting from using bounding hyper-boxes.

The experiments in Table 2 are performed using 100,000 uniformly distributed data in 2, 4, 8, 16, and 32 dimensions. The node size was kept fixed at 8192

bytes, hence the leaf fan-out (LFO) and internal node fan-out (IFO) decreases as the dimension increases. The ratio between the largest and smallest eigenvalues was fixed at 4 for all the experiments.

Table 2. A comparison between the transformation and lower-bound method for 100,000 uniformly distributed vectors (ratio used: 4, bucket size: 8192 bytes).

			Bounding (hyper)box			Bounding (hyper)sphere			Faloutsos method		
dimension	LFO	IFO	ltouch	lused	inode	ltouch	lused	inode	ltouch	lused	inode
2	682	255	1.4	1.4	1	1.4	1.4	1	3.2	2.7	2.0
4	409	146	4.5	2.7	2.2	3.6	2.4	2.3	12.5	5.5	4.5
8	227	78	89.5	8.9	4.8	39.5	7.0	4.0	175.1	17.1	10.4
16	120	40	740.6	25.9	20.4	712.5	25.9	22.0	1489.2	46.0	43.8
32	62	20	1613.0	41.0	87.0	1613	42.0	87.0	3226.0	88.0	174.0

In Table 2, the number of nodes accessed are much higher in high dimensions. This is partly due to the smaller internal/leaf node fan-outs (we kept the node size fixed for all dimensions). Similar to the experimental results using texture vectors, the number of leaf nodes and internal nodes used by our method (*lused* and *inodes*) is approximately half of the number used by Faloutsos' method. In terms of the number of leaf nodes accessed (*ltouch*), our method consistently accessed less than 50% of the number accessed by Faloutsos' method when the dimension is greater than four.

The result experimental results using bounding spheres (up to dimension 16) further support the fact that the query sensitive area resulting from using bounding hyper-spheres will get smaller than the one resulting from using bounding hyper-boxes. But the experimental results in 32 dimension shows that the performance is almost the same for both method. This is due to the inherent problem with hierarchical tree access method in high dimension. In this case, the search actually has degraded into linear search (the variations are due to the randomization used in the experiments).

The experiments are verified by comparing the results of the k nearest neighbour search with the results obtained via simple linear search and comparing the results of the two methods.

6 Conclusion

Nearest-neighbour search on multi-dimensional data is an important operation in many areas including multimedia databases. The similarity measurement usually involves weighting on some of the dimensions and weighting may vary between queries. Existing tree structured spatial access methods cannot support this requirement directly. We have developed an algorithm to support variable

distance metrics in queries. The algorithm uses the bounding hyper-box of the k-nn hyper-sphere instead of calculating intersections directly. It has $O(d^3)$ complexity. We derive the theoretical foundation of the algorithm and give a detailed implementation in the 2D case. We also extend the algorithm to higher dimensions and give a heuristic algorithm to detect the intersection between the query disk and the bounding envelopes of the tree nodes. We provide analytical and empirical results regarding the performance of our approach in terms of the number of disk pages touched. The experiments go up to 32 dimensions. The algorithm has a significant impact on k-nn search and various index trees. There are also research issues in speeding up the transformation. As we can see from the derivation of the algorithm, the major computational cost is in cross-space transformation.

References

1. David W. Aha. A study of instance-based algorithms for supervised learning tasks: Mathematical, empirical, and psychological evaluations (dissertation). Technical Report ICS-TR-90-42, University of California, Irvine, Department of Information and Computer Science, November 1990. 64, 65
2. S. Belkasim, M. Shridhar, and M. Ahmadi. Pattern classification using an efficient KNNR. *Pattern Recognition*, 25(10):1269–1274, 1992. 64
3. Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975. 65
4. Christos Faloutsos. *Searching Multimedia Databases by Content*. Advances in Database Systems. Kluwer Academic Publishers, Boston, August 1996. 65, 66
5. Christos Faloutsos, William Equitz, Myron Flickner, Wayne Niblack, Dragutin Petkovic, and Ron Barber. Efficient and effective querying by image content. *J. of Intelligent Information Systems*, 3:231–262, July 1994. 66
6. Christos Faloutsos, M. Ranganathan, and Yannis Manolopoulos. Fast subsequence matching in time-series databases. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 419–429, May 1994. 64
7. Myron Flickner, Harpreet Sawhney, Wayne Niblack, Jonathan Ashley, Qian Huang, Bryan Dom, Monika Gorkani, Jim Hafner, Denis Lee, Dragutin Petkovic, David Steele, and Peter Yanker. Query by image and video content: The QBIC system. *IEEE Computer*, pages 23–32, September 1995. 64
8. Jerome H. Friedman, Jon Louis Bentley, and R.A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. on Math. Software (TOMS)*, 3(3):209–226, September 1977. 65
9. Keinosuke Fukunaga and Larry D. Hostetler. Optimization of k-nearest-neighbor density estimates. *IEEE Transactions on Information Theory*, IT-19(3):316–326, May 1973. 64
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, MA, June 1984. 65
11. Gisli R. Hjaltason and Hanan Samet. Ranking in spatial databases. In Max J. Egenhofer and John R. Herring, editors, *Advances in Spatial Databases, 4th International Symposium, SSD'95*, volume 951 of *Lecture Notes in Computer Science*, pages 83–95, Berlin, 1995. Springer-Verlag. 65

12. Jesse Jin, Lai Sin Tiu, and Sai Wah Stephen Tam. Partial image retrieval in multimedia databases. In *Proceedings of Image and Vision Computing New Zealand*, pages 179–184, Christchurch, 1995. Industrial Research Ltd. 64, 73
13. Jesse S. Jin, Guangyu Xu, and Ruth Kurniawati. A scheme for intelligent image retrieval in multimedia databases. *Journal of Visual Communication and Image Representation*, 7(4):369–377, 1996. 64
14. D. Kibler, D. W. Aha, and M. Albert. Instance-based prediction of real-valued attributes. *Computational Intelligence*, 5:51–57, 1989. 64, 65
15. Flip Korn, Nikolaos Sidiropoulos, Christos Faloutsos, and Eliot Siegel. Fast nearest-neighbor search in medical image databases. In *International Conference on Very Large Data Bases*, Bombay, India, Sep 1996. 66
16. Ruth Kurniawati, Jesse S. Jin, and John A. Shepherd. The SS$^+$-tree: An improved index structure for similarity searches in a high-dimensional feature space. In *Proceedings of the SPIE: Storage and Retrieval for Image and Video Databases V*, volume 3022, pages 110–120, San Jose, CA, February 1997. 65, 65, 72, 72
17. Ruth Kurniawati, Jesse S. Jin, and John A. Shepherd. Efficient nearest-neighbour searches using weighted euclidean metrics. Technical report, Information Engineering Department, School of Computer Science and Engineering, University of New South Wales, Sydney 2052, January 1998. 69, 73
18. Nick Roussopoulos, Stephen Kelley, and Frédéric Vincent. Nearest neighbor queries. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 71–79, San Jose, California, May 1995. 65, 73
19. Robert F. Sproull. Refinements to nearest-neighbour searching in k-dimensional trees. *Algorithmica*, 6:579–589, 1991. 65
20. Gilbert Strang. *Introduction to applied mathematics.* Wellesley-Cambridge Press, Wellesley, MA, 1986. 68
21. Gilbert Strang. *Linear algebra and its applications.* Harcourt, Brace, Jovanovich, Publishers, San Diego, 1988. 69
22. David A. White and Ramesh Jain. Similarity indexing with the SS-tree. In *Proc. 12th IEEE International Conference on Data Engineering*, New Orleans, Louisiana, February 1996. 65

The Acoi Algebra: A Query Algebra for Image Retrieval Systems

Niels Nes and Martin Kersten

University of Amsterdam
{niels,mk}@wins.uva.nl

Abstract. Content-based image retrieval systems rely on a query-by-example technique often using a limited set of global image features. This leads to a rather coarse-grain approach to locate images. The next step is to concentrate on queries over spatial relations amongst objects within the images. This calls for a small collection of image retrieval primitives to form the basis of an image retrieval system. The Acoi algebra is such an extensible framework built on the relational algebra. New primitives can be added readily, including user-defined metric functions for searching. We illustrate the expressive power of the query scheme using a concise functional benchmark for querying image databases.

keywords: Image Retrieval, Query Algebra, Features, and IDB.

1 Introduction

With the advent of large image databases becoming readily available for inspection and browsing, it becomes mandatory to improve image database query support beyond the classical textual annotation and domain specific solutions, e.g. [21]. An ideal image DBMS would provide a data model to describe the image domain features, a general technique to segment images into meaningful units and provide a query language to study domain specific algorithms with respect to their precision and recall capabilities. However, it is still largely unknown how to construct such a generic image database system.

Prototype image database systems, such as QBIC [6], WebSeek[18], and PictoSeek [8] have demonstrated some success in supporting domain-independent queries using global image properties. Their approach to query formulation (after restricting the search using textual categories) is based on presenting a small sample of random images taken from the target set and to enable the user to express the query (Q_1) "find me images similar to this one" by clicking one of the images provided. Subsequently the DBMS locates all images using its built-in metrics over the global color distribution, texture, or shape sets maintained.

However, this evaluation technique is bound to fail in the long run for several reasons. First, random sample sets to steer the query process works under the assumption that there is a clear relationship between color, texture and shape

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 77–88, 1998.

and the semantic meaning. This pre-supposes rather small topical image databases and fails when the database becomes large or filled from many sources, such as envisioned for the Acoi image database[1].

Second, the global image properties alone are not sufficient to prune false hits. Image databases are rarely used to answer the query Q_1. Instead, the intended user query (Q_2) is :"find me an image that contains (part of) the one selected" where the containment relationship is expressed as a (predefined) metric over selected spatial and image features or directly (Q_3) :" find me an image that contains specific features or objects using my own metric". In addition to color distribution and texture, spatial information about object locality embedded in the image is needed. A prototype system that addresses these issues is VisualSEEK[18] graphical user interface for the WebSeek image retrieval system.

What are the necessary primitives to express the metric? For example, in a large image database one could be interested to locate all images that contain part of the Coca-Cola logo. This query could be formulated by clipping part of a sample Coca-Cola logo to derive its reddish (R) and white (W) color and to formulate a query of the form:

select display(i)
from image_region r1,r2, image i
where distance(r1.avghue, R) < 0.2
and distance(r2.avghue, W) < 0.2
and r1 **overlaps** r2
and r1,r2 **in** i
sort by distance(r1.avghue, R), distance(r2.avghue, W)

This query uses two primitive parameterized metric functions. The function *distance* calculates a distance in the color space and *overlaps* determines region containment. The former is defined as part of the **color** data type and the latter for the **region** data type.

A challenge for image database designers is to identify the minimal set of features, topological operators, and indexing structures to accommodate such image retrieval queries. In particular, those (indexed) features where their derivation from the source image is time consuming, but still can be pre-calculated and kept at reasonable storage cost. This problem becomes even more acute when the envisioned database is to contain over a million images.

In [13] we introduced an extensible image indexing algorithm based on rectangular segmentation of regions. Regions are formed using similarity measures. In this paper we extended this approach with a query algebra to express queries over the image database.

For such an image retrieval algebra we see three global requirements.

1. The algebra should be based on an extensional relational framework.
2. The algebra should support proximity queries and the computational approach should be configurable by the user.

[1] Acoi is the experimental base for the national project on multi-media indexing and search (SION-AMIS project), http://www.cwi.nl/~acoi/Amis

3. The algebra should be computationally complete to satisfy the wide community of (none-database) image users.

The remainder of this report is organized as follows. In Section 2 we explain our database model, review available region representations and query primitives for image retrieval systems. Also we provide a short introduction to our underlying database system, called Monet. Section 3 explains the query primitives. In Section 4 we define the Acoi Image Retrieval Benchmark, which is the basis for the experimentation reported in section 5. We conclude with an indication of future research topics.

2 Image Databases

This Section introduces the data model for query formulation. The data model is based on regions as the abstraction of the image segmentation process. In section 2.2 we review several region representation methods. In section 2.3 query language extensions for image retrieval are reviewed to provide the background information and to identify the requirements imposed on the image DBMS. Since our image retrieval system is built using the Monet database system we also give a short introduction to Monet.

2.1 Image Database Model

The Acoi database is described by the ODL specification shown in Figure 1.

```
interface Img {
        relationship set < Pix > data inverse Pix::image;
};
interface Pix {
        relationship Img image inverse Img::data;
        relationship Reg region inverse Reg::pixels;
};
interface Reg {
        relationship set < Pix > pixels inverse Pix::region;
        relationship set < Seg > segments inverse Seg::regions;
};
interface Seg {
        relationship set < Reg > regions inverse Reg::segment;
        relationship set < Obj > object inverse Obj::segments;
};
interface Obj {
        relationship set < Seg > segments inverse Seg::object;
};
```

Fig. 1. Data Model

The *data* relationship relates raw pixel information with an image. This is a virtual class, because each pixel is accessed from the image representation upon

need. The *region* relationship expresses that each pixel is part of one region only. For the time being we use rectangular regions to simplify implementation and to improve performance. The architecture has been set up to accommandate other (ir-) regular regions, like hexagons and triangulation, as well.

The *segments* mapping combines regions into meaningful units. The segments are typically the result of the image segmentation process, which determines when regions from a semantic view should be considered together. The model does not prescribe that regions are uniquely allocated to segments. A region could be part of several segments and applying different segmentation algorithms may result in identical region sets. The segmentation algorithm used for the current study is based on glueing together regions based on their average color similarity and physical adjacency, details of which can be found in [13].

The relationship object of the segments interface, expresses that segments can form a semantically meaningful object. An example is a set of segments together representing a car.

2.2 Segment Representation

The bulk of the storage deals with region representation, for which many different approaches exist. All have proven to be useful in a specific context, but none is globally perfect. The chain code as described by Freeman [7] encodes the contour of a region using the 8-connected neighborhood directions. Chain codes are used in edge, curve and corner finding algorithms [11]. It is not useful for region feature extraction, since it only represents part of the boundary of an area, no interior. The complexity is $O(p)$ for both storage and performance, where p is the perimeter of the region.

Many boundary representations exist [10], e.g. polygons and functional shape descriptors. Functional shape descriptors use a function to approximate the region boundary. Fourier, fractal and wavelet analysis have been proposed for this [3,12,17]. Although these representations have very low storage requirements, i.e. each boundary is represented using a few parameters, they are of limited use aside from shape representation. Recalculation of the regions interior from polygons is very hard and from functional descriptions generally impossible.

Another representation to describe the interior of the region is run length encoding using (position, length) pairs in the scan direction [9]. Diagonal shaped regions are handled poorly by this coding schema.

The pyramid structures [20,19] represent an region using multiple levels of detail. They are used in image segmentation and object recognition [20,15]. These structures are very similar to the quad tree [16]. The quad tree is a hierarchical representation, which divides regions recursively into four equal sized elements. The complexity of this structure per region is $O(p + n)$, where the region is located in a $2^n * 2^n$ image and p is again the perimeter of the region. Quad trees can be stored efficiently using a pointerless representation. The quad tree has been used to represent binary images efficiently. The tree needs only to store those regions which have a different color than its parent nodes.

Since none of the structures above solve the regions representation problem, there is a strong need for an extensible framework. It would permit domain

specific representations to be integrated into a database kernel, such that scalable image databases and their querying becomes feasible.

To explore this route we use a minimalistic approach, i.e. regions are described by rectangular grids. The underlying DBMS can deal with them in an efficient manner. The domain specific rules and heuristics are initially handled by the query language and its optimizer.

2.3 Image Retrieval Algebra

The image retrieval problem is a special case of the general problem of object recognition. When objects can be automatically recognized and condensed into semantic object descriptors, the image retrieval problem becomes trivial. Unfortunately, object recognition is only solved for limited domains. This calls for an image feature database and a query algebra in which a user can express domain specific knowledge to recognize the objects of interest.

Research on image retrieval algebras has so far been rather limited. The running image retrieval systems support query by example[6] or by sketch [18], only. For example, the interface of the QBIC system lets the user choose for retrieval based on keywords or image features. These systems have a canned query for which only a few parameters can be adjusted. It does not provide a functional or algebraic abstraction to enable the user to formulate a specific request. In the WebSeek Internet demo the user can adjust a color histogram of a sample image to specify the more important colors. However, this interface allows no user defined metric on colors.

Only Photobook [14] allows for user defined similarity metric functions through dynamically loadable C-libraries. Although this approach is a step forward, it is still far from a concise algebraic framework that has boosted database systems in the administrative domain. In section 3 we introduce the components of such an algebra.

2.4 Extensible Database Systems

Our implementation efforts to realize an image database system are focussed on Monet. Monet has been designed as a next generation system, anticipating market trends in database server technology. It relies on a network of workstations with affordable large main memories ($>$ 128 MB) per processor and high-performance processors ($>$ 50 MIPS). These hardware trends pose new rules to computer software – and to database systems – as to what algorithms are efficient. Another trend has been the evolution of operating system functionality towards micro-kernels, i.e. those that make part of the Operating System functionality accessible to customized applications.

Given this background, Monet was designed along the following ideas:

– *Binary relation storage model.* Monet vertically partitions all multi-attribute relationships in Binary Association Tables (BATs), consisting of `[OID,attribute]` pairs. This Decomposed Storage Model (DSM) [5] facilitates table evolution. And it provides a canonical representation for a variety

of data models, including an object-oriented model [1]. Moreover, it leads to a simplified database kernel implementation, which enables readily inclusion of additional data types, storage representations, and search accelerators.

– *Main memory algorithms.* Monet makes aggressive use of main memory by assuming that the database hot-set fits into its main memory. For large databases, Monet relies on virtual memory management by mapping files into it. This way Monet avoids introducing code to 'improve' or 'replace' the operating system facilities for memory/buffer management. Instead, it gives advice to the lower level OS-primitives on the intended behavior[2] and lets the MMU do the job in hardware. Experiments in the area of Geographical Information Systems[2] and large object-oriented applications [1] have confirmed that this approach is performance-wise justified.
– *Monet's extensible algebra.* Monet's Interface Language (MIL) is an interpreted algebraic language to manipulate the BATs. In line with extensible database systems, such as Postgres, Jasmine and Starburst, Monet provides a Monet Extension Language (MEL). MEL allows you to specify extension modules to contain specifications of new atomic types, new instance- or set-primitives and new search accelerators. Implementations have to be supplied in C/C++ compliant object code.

3 Algebraic Primitives

Analysis of the requirements encountered in image retrieval and the techniques applied in prototype image systems, such as [6,18,8], indicate the need for algebraic operators listed in Table 1. The parameter i denotes an image, p a pixel, r a region, s a segment and o an object. Most functions are overloaded as indicated by a combination of $iprso$.

The first group provides access to the basic features of images, pixels, regions, segments and objects. Their value is either stored or calculated upon need. The Point, Color, Vector and Histogram datatypes are sufficient extensions to the base types supported by the database management system to accommodate the features encountered in practice so far.

The second group defines topological relationships. This set is taken from [4], because there is no fundamental difference between spatial information derived from images and spatial information derived from geographic information systems.

The third group addresses the prime algorithmic steps encountered in algorithms developed in the Image processing community. They have been generalized from the instance-at-a-time behavior to the more convenient set-at-a-time behavior in the database context. This group differs from traditional relational algebra in stressing the need for θ-like joins and predicates described by complex mathematical formulae.

[2] This functionality is achieved with e.g. `mmap()`, `madvise()`,and `mlock()` Unix system calls.

Properties	
$area(iprso)$	$\rightarrow float$
$perimeter(iprso)$	$\rightarrow float$
$center(iprso)$	$\rightarrow point$
$avg\_color(iprso)$	$\rightarrow color$
$color\_hist(iprso)$	$\rightarrow Histogram$
$texture(iprso)$	$\rightarrow vector$
$moment(iprso)$	$\rightarrow float$
Topological operations	
$touch(prso, prso)$	$\rightarrow boolean$
$inside(prso, prso)$	$\rightarrow boolean$
$cross(prso, prso)$	$\rightarrow boolean$
$overlap(prso, prso)$	$\rightarrow boolean$
$disjoint(prso, prso)$	$\rightarrow boolean$
Join operations	
$F\_join_{f(prso,prso)}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
$M\_join_{d(prso,prso),m}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
$P\_join_{p(prso,prso)}(\{prso\}, \{prso\})$	$\rightarrow \{prso\}$
Selection operations	
$F\_find_{f(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow iprso$
$M\_select_{d(iprso,iprso),m}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
$P\_select_{p(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
Ranking and Sample operations	
$P\_sort(\{iprso\})$	$\rightarrow \{iprso\}$
$M\_sort_{d(iprso,iprso)}(\{iprso\}, iprso)$	$\rightarrow \{iprso\}$
$N\_sort(\{iprso\})$	$\rightarrow \{iprso\}$
$Top(\{iprso\}, int)$	$\rightarrow \{iprso\}$
$Slice(\{iprso\}, int, int)$	$\rightarrow \{iprso\}$
$Sample(\{iprso\}, int)$	$\rightarrow \{iprso\}$

Table 1. The Image Retrieval Algebra

A $fitness$ join (F_join) combines region pairs maximizing a fitness function, $f(rs, rs) \rightarrow float$. The pairs found merge into a single segment. The *metric* join (M_join) finds all pairs for which the distance is less than the given maximum m. The distance is calculated using a given metric function, $d(rs, rs) \rightarrow float$. The last function in this group, called *predicate* join (P_join), is a normal join which merges regions for which the predicate p holds. An example of such an expression is the predicate "similar", which holds if regions r_1 and r_2 touch and the average colors are no more than 0.1 apart in the domain of the color space. A functional description is:

$$\begin{aligned}&\text{similar}(r\ r_1, r\ r_2) :=\\ &\qquad touch(r_1, r_2) \text{ and}\\ &\qquad distance(r_1.avg\_color, r_2.avg\_color) < 0.1\end{aligned}$$

The next group of primitives is needed for selection. The fitness find (F_find) returns the region which fits best to the given region, according to fitness function $f(rs, rs)$. The metric select (M_select) returns a set of regions at most at distance m, using the given metric $d(rs, rs)$ function. The predicate select (P_select) selects all regions from the input set for which the predicate is valid.

Join operations	result
$F\_join_f(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR, / l' \exists r' \in LR \wedge f(l',r') > f(l,r)\}$
$M\_join_{d,m}(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR \wedge d(l,r) < m\}$
$P\_join_p(L,R) \rightarrow \{prso\}$	$\{lr \mid lr \in LR \wedge p(l,r)\}$
Selection operations	**result**
$F\_find_f(L,r) \rightarrow iprso$	$l \in L, / l' \exists \in L \wedge f(l',r) > f(l,r)$
$M\_select_{d,m}(L,r) \rightarrow \{iprso\}$	$\{l \mid l \in L \wedge d(l,r) < m\}$
$P\_select_p(L,r) \rightarrow \{iprso\}$	$\{l \mid l \in L \wedge p(l,r)\}$

Table 2. Signatures of the Join and Selection operations

The last group can be used to sort region sets. We have encountered many algorithms with a need for a partial order. P_sort derives a partial order amongst objects. Each entry may come with a weight which can be used by the *metric* sort (M_sort). This sort operation is based on a distance metric between all regions in the set and a given region. The N_sort uses a function to map regions onto the domain $\mathcal{N}$.

After the partial order the Top returns the top n objects of the ordered table. The *Slice* primitive will slice a part out of such an ordered table. The *Sample* primitive returns a random sample from the input set.

4 Acoi Image Retrieval Benchmark

The next step taken was to formulate a functional benchmark of image retrieval problems. Many such performance benchmarks exist for DBMS for a variety of application areas. Examples in transaction processing are the TP series (TPC-C and TPC-D) and in geographic information systems the SEQUOIA 2000 storage benchmark. We are not aware of similar benchmarks for image retrieval. The construction of such a benchmark is one of the goals of Amis. Both the database and image processing community would benefit from such a public accessible benchmark.[3]

Its function is to demonstrate and support research in image processing and analysis in a database context. Therefore, we derived the following characteristics from the algorithms used in the image processing domain.

- *Large Data Objects* The algorithms use large data objects. Both in terms of base storage (pixels), but also the derived data incurs large space overhead.

[3] Readers can contact authors for a copy of the Acoi Benchmark.

- *Complex Data Types* The algorithms use specialized complex data types. Derived data is often stored in special data structures.
- *Fuzzy data* The computational model used is based on heuristics and fuzzy data. This fuzzy data should be accompanied by some form of fuzzy logic.

The Acoi Benchmark Data The data for the benchmark consists of two Image sets, one of 1K images and one of 1M images. The images are retrieved randomly from the Internet using a Web robot. The set contains all kinds of images, i.e. binary and gray scale, small and large but mostly color images.

The Acoi Benchmark Queries Based on the characteristics encountered in the image processing community a set of 6 distinctive queries for the benchmark was identified, which are shown in Table 3.

Query 1 loads the database DB from external storage. This means storing images in database format and calculation of derived data. Since the benchmark involves both global and local image features this query may also segment the images and pre-calculate local image features.

Query 2 is an example of global feature extraction as used in QBIC. This query extracts a normalized color histogram. We only use the Hue component of the Hue, Saturation, Intensity color model. The histogram has a fixed number of 64 bins. In query 3 these histograms are used to retrieve histograms within a given distance and the related images. The histogram h should have 16 none-zero bins and 48 zero. The none-zero bins should be distributed homogeneous over the histogram. The query Q3a sorts the resulting set for inspection.

Query 4 finds the nearest neighboring regions in an image. Near is defined here using a user-defined function, f. This function should be chosen so that neighbors touch and that the colors are as close as possible.

Query 5 segments an input image. Segmentation can also be done with specialized image processing functions, but to show the expressive power of the algebra we also include it here in its bare form. Finally Q6 searches for all images in the database which have similar segments as the example image. The resulting list of images is sorted in query 6a.

The Benchmark Evaluation To compare the results of various implementations of the benchmark we used the following simple overall evaluation scheme. The performance of the Acoi Benchmark against different implementation strategies can be compared using the sum of all query execution times. This way moving a lot of pre-calculation to the DB-load query will not improve performance unless the information stored has low storage overhead and is expensive to recalculate on the fly.

5 Performance Assessment

The benchmark has been implemented in Monet using its extensible features. Details about Monet can be found at {http://www.cwi.nl/∼monet}. The DB-load

nr	query
Q1	DB-load
Q2	$\{h \mid i \in Ims \wedge h = normalized\_color\_histogram(i)\}$
Q3	$\{i \mid i \in Ims \wedge L^2 distance(normalized\_color\_histogram(i), h) < 0.1\}$
Q3a	sort Q3
Q4	$\{n_1 n_2 \mid n_1 n_2 \in Regs(im) \wedge \ / n3 \in Regs f(n_1, n_3) > f(n_1, n_2)\}$
Q5	$\{rs \mid rs \subset Regs(i) \wedge \forall r_1 r_2 \in RS :$ $L^2 distance(avg\_color(r_1), avg\_color(r_2)) < 0.1$ $\wedge \exists s_0 \ldots s_n \in rs :$ r touch $s_0 \wedge$ s_i touch $s_{i+1} \wedge$ s_n touch $s\}$
Q6	$\{i \mid \forall s_i \in Q6(i) \exists s_e \in Segs(e)$ $d(s_i, s_e) < min\_dist\}$
Q6a	sort Q6

Table 3. Benchmark Queries

query loads the images using the image import statement into the Acoi_Images set. We only load the images in the system. No pre-calculation has been performed.

The color histogram query (Q2) can be expressed in the Acoi algebra as follows:

```
var Q2 := [normalized_color_histogram](Acoi_Images);
```

The brackets will perform the operation normalized_color_histogram on all images int the Acoi_Images set. It returns a set of a histograms. Q3 uses a M_select with the L^2 metric. The sorting of Q3a can be done using the M_sort primitive. Query Q4 is implemented in the Acoi algebra using a F_join with the function $f(r_1, r_2)$ defined as follows:

$f(r_1, R_2) :=$
dist($r_1.color()$, $r_2.color()$) if r_1.touch(r_2)
max_dist

The queries 5 and 6 are implemented by longer pieces of Monet code. The segmentation of query Q5 use an iterative process. This process can make use of the F_join primitive to find the best touching regions based on the color distance, see [13] for full details.

Query Q6 can be solved using a series of M_select calls. For each segment in the example image we should select all Images with similar segments, where similar is defined using the metric given. The intersection of the selected images is the result of query 6. This can be sorted using the M_sort primitive.

The Benchmark Results We run these queries using the small Acoi database of 1K images. The small benchmark fits in main memory of a large workstation. The database size is approximately 1G. We used a sparc ultra II with 128 MB of

main memory running the Solaris operating system, to perform the benchmark on. Using the Acoi algebra we could implement the benchmark with very little effort.

The initial results can be found in Table 5. Overall the benchmark took 3468.8 seconds.

In the result we can see that the DB-load query takes more than 80 percent of the overall benchmark result. This unexpected result stems from heavy swapping of the virtual memory management system. Main memory runs out quickly, so swapping will influence the performance. Based on our early experimentation with multi-giga-byte databases this problem can be resolved with some careful loading scripts.

We found that the results of queries Q4 and Q5 were low. The none-optimized current implementation of F_join was responsible for the low performance. To improve it we moved the spatial constrains out of the F_join. This allows us to find candidate pairs based on the spatial relation between regions quickly. This way we improved the performance of the queries Q4 from 5 to 1 second and Q5 from 21 to 1.2 seconds using a few minutes of programming. A similar step in a traditional image programming environment would have meant partly re-coding several pages of c/c++ code.

Query	Time(s)	Query	Time(s)
Q1	2865	Q4	1.0
Q2	598	Q5	1.2
Q3	1.5	Q6	1.5
Q3a	0.3	Q6a	0.3

Table 4. The Acoi Benchmark Results

6 Conclusions

In this paper we introduced an algebraic framework to express queries on images, pixels, regions, segments and objects. We showed the expressive power of the Acoi algebra using a representative set of queries in the image retrieval domain. The algebra allows for user defined metric functions and similarity functions, which can be used to join, select and sort regions. The algebra is extensible with new region properties to accommodate end user driven image analysis in a database context.

We have implemented the algebra within an extensible DBMS and developed a functional benchmark to assess its performance. In the near future we expect further improvement using extensibility in search methods and index structures to improve the performance of the algebra. As soon as the full Acoi database is ready we will perform the benchmark on the set of 1M images.

References

1. P.A. Boncz and M.L. Kwakkel, F. Kersten. High Performance support for OO traversals in Monet. In *BNCOD proceedings*, 1996. 82, 82

2. Peter A. Boncz, Wilko Quak, and Martin L. Kersten. Monet and its Geographic Extensions: a novel Approach to High Performance GIS Processing. In *EDBT proceedings*, 1996. 82
3. R. Chellappa and R. Bagdazian. Fourier Coding of Image Boundaries. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:102–105, 1984. 80
4. E. Clementini, P. Felice Di, and P. Oosterom van. A Small Set of Formal Topological Relationships Suitable for End-user Interaction. In *SSD: Advances in Spatial Databases*. LNCS, Springer-Verlag, 1993. 82
5. G. Copeland and S. Khoshafian. A Decomposed Storage Model. In *Proc. ACM SIGMOD Conf.*, page 268, Austin, TX, May 1985. 81
6. C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and Effective Querying by Image Content. *Intelligent Information Systems 3*, pages 231–262, 1994. 77, 81, 82
7. H. Freeman. On the encoding of arbitrary geometric configurations. *Transactions on electronic computers*, 10:260–268, jun 1961. 80
8. T. Gevers and A. W. M. Smeulders. Evaluating Color and Shape Invariant Image Indexing for Consumer Photography. In *Proc. of the First International Conference on Visual Information Systems*, pages 293–302, 1996. 77, 82
9. S W Golomb. Run-Length Encodings. *IEEE Transactions on Information Theory 12(3)*, pages 399–401, july 1966. 80
10. Anil K. Jain. *Fundamentals of Digital Image Processing*. Prentice-Hall, Englewood Cliffs, NJ, 1989. 80
11. Hong-Chih Liu and M. D Srinath. Corner Detection from Chain-Code. *Pattern Recognition(1-2), 1990*, 23:51–68, 1990. 80
12. B. B. Mandelbrot. *The Fractal Geometry of Nature*. W.H. Freeman and Co., New York, rev 1983. 80
13. N.J. Nes and M.L. Kersten. Region-based indexing in an image database. *In proceedings of The International Conference on Imaging Science, Systems, and Technology, Las Vegas*, pages 207–215, June 1997. 78, 80, 86
14. A. Pentland, R. W. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases. In *SPIE Storage and Retrieval for Image and Video Databases II, No. 2185*, pages 34–47, 1994. 81
15. E. M. Riseman and M. A. Arbib. Computational Techniques in the Visual Segmentation of Static Scenes. *Computer Graphics and Image Processing*, 6(3):221–276, June 1977. 80
16. H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison Wesley, 1990. 80
17. J. Segman and Y. Y. Zeevi. Spherical wavelets and their applications to image representation. *Journal of Visual Communication and Image Representation*, 4(3):263–70, 1993. 80
18. John R. Smith and Shih-Fu Chang. Tools and Techniques for Color Image Retrieval. In *SPIE Storage and Retrieval for Image and Video Databases IV, No 2670*, 1996. 77, 78, 81, 82
19. S. L. Tanimoto and T. Pavlidis. A Hierarchical Data Structure for Picture Processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975. 80
20. L. Uhr. Layered recognition cone networks that preprocess, classify, and describe. *IEEE Transactions on Computers*, 21:758–768, 1972. 80, 80
21. Aref. W.G., Barbara D., and D. Lopresti. Ink as a First-Class Datatype in Multimedia Databases. Multimedia Database Systems, pages 113–160, 1996. 77

A Meta-Structure for Supporting Multimedia Editing in Object-Oriented Databases ⋆

Greg Speegle[1], Xiaojun Wang[2], and Le Gruenwald[3]

[1] Baylor University, Waco, TX 76798, USA
[2] Intervoice, Inc, Dallas TX 75252, USA
[3] The University of Oklahoma Norman, OK 73019, USA

Abstract. Multimedia databases include many types of new data. One common property of these data items is that they are very large. We exploit the concept of transformational representations to logically store images without the bitmap. This technique is similar to views in traditional database systems. The technique is useful when users are editing images in the database to create new images. Our method produces significant space savings over *already compressed* images, and potentially greater savings for video and audio. The method requires the support of multimedia editors. This paper emphasizes the meta-structure needed by the database to support multimedia editing.

1 Introduction

Multimedia has become the quintessential element in computer applications. Programs are purchased on CDROMs loaded with graphics, audio, animations, and videos. Media viewers are common on home computers. Millions of users are putting digital media on the web.

As the demand for multimedia continues to increase, computer systems will have to address two large problems associated with the wide spread use of digital media. The first is that current systems simply store such media as compressed files on disks without providing any organization. This, therefore, requires a tremendous amount of effort from application programmers to interpret and manipulate those files. The second is that even with advanced compression techniques and several gigabyte disks being common, the size of media data causes it to consume enormous amounts of space. For example, a 75 minute CD may use 100 MB of storage, and each frame of a video may use over 1 MB [8]. This problem is increased when users edit digital media in order to refine their presentations.

Editing can occur in video games (graphics), recording studios (sound effects), and video-on-demand systems (editing for content). In each of these examples, the application must maintain many large multimedia objects that are similar. This causes a tremendous amount of replicated information to be stored. This situation would arise in any multimedia authoring application in

⋆ Work partially supported by Baylor University Grant 009-F95

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 89–102, 1998.

which users will want to save each version of a file. The storage cost of such redundant information can vary from 2KB for a small graphic to over a gigabyte for a movie. Without a technique to efficiently store the redundant data, the systems supporting such applications would quickly run out of memory.

We propose an object-oriented MultiMedia Database Management System (MMDBMS) to solve the problems of organization and space consumption. Instead of storing the objects themselves, our MMDBMS stores the editing operations, called the transformational representations, used to create media objects. The collection of all information used to make a new image is called a *transform*. Since the editing operations would only require a few bytes of space, transforms would significantly reduce storage requirements for already compressed data [11,12,1].

Transformational representation of information is called a *view* in relational databases, and an *intensional database relation* in logical models like datalog [14]. Within media editing tools, transformational representations of objects are also used. For example, some editors, such as JPEGview [3], can store an object by copying the original file and then appending to it a series of commands which indicate how the edited image should be recreated. Unfortunately, using application specific transformational representations does not allow an image to be correctly viewed by another editor. This happens because the editing commands used by one system are meaningless to another. Likewise, this method has the disadvantage of storing redundant data, since the original file is copied to the edited one. Although in a file system this redundancy is unavoidable – since it is impossible to prevent the deletion or movement of files – in a database, actions can be performed as part of a deletion operation in order to protect the derived data. One simple solution is to instantiate the derived data before allowing the deletion to complete. Thus, we do not need the redundant storage. Furthermore, image degradation due to repeated application of lossy compression algorithms is eliminated. Thus, better image quality and greater space efficiency can be gained.

Ideas similar to multimedia transforms can be found in [2] and [13]. In [2], multimedia objects are treated as binary large objects (BLOBs). Editing of media objects is translated into operations on BLOBs. Handles to BLOBs are stored in a relational database. Unfortunately, BLOB operations are not sufficient to perform all required media object operations. For example, there is no way to describe the dithering needed to double the size of an image. In [13], a notion similar to transforms is presented in which scripts are passed to image editing tools in order to create new images. However, their approach performs these operations outside the database itself. This can lead to problems with deleted references and poor specifications, both of which can be solved by including the new images within the database.

Within the OODBMS literature, a similar notion to our approach is the concept of versioning [9]. Versions allow modifications to an object without losing the previous state. This is accomplished by having additional structures, similar in nature to those proposed here, which keep up with the history of the object.

However, there is a fundamental difference between versions and our work. In our system, each modification creates a totally independent object, not merely a copy of another object. Thus, the support structure for our work must be different in terms of modifications to the database.

The rest of the paper is organized as follows. Section 2 presents the basic model used by our meta-structure. Section 3 presents implementation details for capturing the operations, storing them in the database, and instantiating the transforms. Section 4 concludes this work and outlines the rest of our project. It should be noted that this paper focuses on images. However, we intend to apply this meta-structure to all media types.

2 The Basic Model

Our base model for an MMDBMS with transforms contains two distinct parts. The first part, which is the emphasis of this paper, is the meta-structure needed to support transforms. This is covered in Section 2.1. A second needed component involves the language used to define the operations for the media. Our language is only a small component of this paper, and is simply used to show that such a language is possible. Our proposed language is in Section 2.2.

2.1 Meta-Structure for Views

Our MMDBMS assumes the existence of a media hierarchy – a set of classes which describe the multimedia data. At database design time, the DBA determines if a class is *transformable*. This is a property of the class, and as such it is inherited by its subclasses. If a class is transformable, then by default, all updates to objects in the class will result in the specification of objects, and not create new media objects to be stored in the database. Thus, each of these virtual objects will have a transformational representation of how the object can be recreated from objects physically stored in the database, but they will not include the binary representation of the object.

Note that this implies that specifications are defined on objects, rather than on classes. In traditional databases, views defined like this would be of little benefit, but the extremely large size of individual objects in multimedia systems allows this approach to reduce storage costs. However, using objects as the basis for transforms does require that the system have additional information in order to keep up with the transforms. This additional information is the basis of our meta-structure. We create a *transform-dag* which represents the history of the transform. An arc in a transform-dag from o_1 to o_2 means that o_1 is used to create o_2. In this case, o_1 is called the parent, and o_2 is called the child. As its name suggests, a transform-dag is a directed acyclic graph. The roots of every transform-dag are stored objects. Every transform is a member of exactly one transform-dag. We consider a stored object to be the root of many dags if the dags contain no common transforms. Likewise, each class may have many transform-dags, and a transform-dag may span classes.

Once a class has been defined as transformable, a new system defined attribute is added to the class to support transform-dags. Although it would have a unique system defined name, we use the attribute name *children* in this paper. This attribute would have the type *set-of oids*, where each oid is a transform, and would be a child in the transform-dag.

The system must also create a special *transform-of*-**class** for each transformable class. The *transform-of*-**class** is semi-automatically generated. The *transform-of*-**class** contains exactly the same attributes and methods as the base class, including the children attribute added by the system, except for the following changes:

1. A system defined attribute *parent* is added. The parent is also a *set-of oids*, and represents the objects that were used to create an instance of the *transform-of*-**class**.
2. Two specifications are added. The short specification defines the creation of an object from its parents. The long specification defines the creation of an object from stored objects.
3. A state attribute is added. This is also a system defined attribute which is used to determine the status of transforms. A transform may be either Permanent–meaning a user has saved it; Stable–meaning the system has temporarily saved it; or Transient–meaning the objects can can have their media content modified.
4. The binary representation attribute is removed. This is the part which is semi-automatic, since there is no standard way to describe this field. The binary representation is the physical data which defines a stored object.

An example of a **jpeg** class and a *transform-of*-**jpeg** class is in Figure 1. Note that the *transform-of*-**class** is not a superclass of the **jpeg** class since the specifications are not inherited, nor is it a subclass since the binary representation is no longer present.

Both specifications are linked lists of *Specification Entries,* denoted SE. An SE consists of two parts. The first part is a text description of the command issued by the user. The description consists of two components. The first component is the actual model language command issued, and the second is the parameter required by that command. The second part of the SE is called the *reference* and is used to structure the SE. The reference also consists of two parts, the base and the offset. The base is the object id of a *stored* media object which is one of the roots of the transform-dag of the transform. The offset is an integer value which indicates what changes (if any) have been done to the base object. A negative value (usually -1) indicates that the object has not been edited, and the base object must be retrieved from the database. A value of 0 indicates that the base object has been modified, and the desired image is the result of the previous SE. A positive value is the object id of the SE which represents all of the changes to the base object. Note that the use of 0 is redundant since the oid of the previous SE could also be used. However, this condition is so common, that using a special value to represent it allows an easy optimization of holding the previous SE in memory so that it can be accessed directly.

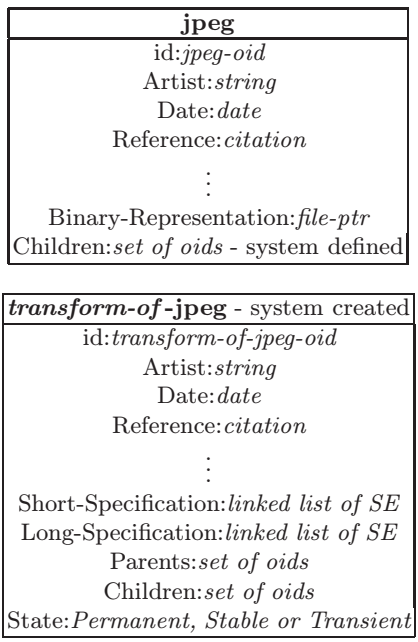


Fig. 1. Example Classes

In order for an MMDBMS to exploit transforms, it must be the case that the editing tools support transforms as well. Some editing tools already use this concept in order to prevent image degradation, but they must go a step further in order to interface with the MMDBMS. The editing tools must be able to transmit to the database system the operations performed by the user. Furthermore, these operations must comply with the logical model language (LML) of the database. An example LML is presented in Section 2.2. It is not practical for the database system to translate proprietary commands into the model language, simply because there are too many different implementations possible. Likewise, the editing tool must transmit to the database system additional operations to retrieve, store, modify and delete images. This is similar to tools accessing an SQL database today. Once the formal model language is established, then it is reasonable for tools to comply with a standard. As image processing continues to mature, such a standard is very likely to emerge, and may even evolve from current standards being developed for handling derived video information, such as MPEG-4 [6].

However, even if media editing matures to the point that this type of functionality can be achieved, a MMDBMS which uses transforms has to fulfill many requirements. Such a system must be able to instantiate objects from formal model specifications, and provide the media to editors on demand. It must be able to accept commands from editors and integrate them into specifications. Finally, it must maintain the transform-dag with media objects being added and being deleted from the system. Section 3 provides details on how these requirements can be fulfilled.

2.2 Model Language for Images

In order for transforms to be defined on multimedia objects, a standard LML must be developed. We predict this language to evolve once editing tools mature. However, in order to continue with our work, we propose a simple image modeling language which captures many of the transformations performed by users with editing tools. Further work is in progress to determine a complete, minimal and independent set of operations for image editing [1].

Our model language for images is based on the image algebra defined in [10]. Ritter's Image Algebra is a mathematical definition of operations used to process and analyze images. As such, it contains a large number of complex operations. This allows the image algebra to capture not only operations performed on images, but also image recognition operations like edge detection.

Our goal is to define a simple set of operations which still capture all of the editing commands performed by users. Thus, for this work we use the five operations: Merge, Define, Mutate, Modify and Combine. These operations form a natural set of basic operations for image manipulations in systems which do not add information to images by techniques such as drawing. Drawing, manipulating the color palette and channel operations are needed in a complete set of image editing tools. Finally, this restricted model is designed only for RGB (red-green-blue) color models.

The Merge operation combines two images together. One image is the base, and the other is the new information. The new information is merged into the base. The Merge operation allows for several different types of combinations. For example, the merge can be transparent, meaning that the base image is displayed wherever the new image is showing its background color, or it can be opaque, meaning that the background information of the new image is also pasted into the base image. The Merge operation works even if the images are not the same size or shape. If the new image overlaps an area not originally part of the base image, the result is simply extended to include the new area.

The Define operation is used to define regions within an image. A series of points defines a polygon within an image. The define operation then either adds the space within the polygon to the defined region, or it removes the polygon from the defined region, or it adds (removes) the image NOT within the polygon to the defined region. Within this work, whenever we refer to an image, the region of an image created with the define operation can also be used.

$$\begin{bmatrix} \cos\theta & -\sin\theta & x_r(1-\cos\theta)+y_r(\sin\theta) \\ \sin\theta & \cos\theta & y_r(1-\cos\theta)-x_r(\sin\theta) \\ 0 & 0 & 1 \end{bmatrix}$$

The Mutation Matrix

$$\begin{bmatrix} 0*\cos 90 - 0*\sin 90 + 25*(1-\cos 90) + 25*\sin 90 \\ 0*\sin 90 + 0*\cos 90 + 25*(1-\cos 90) - 25*\sin 90 \\ 0*0+0*0+1*1 \end{bmatrix} = \begin{bmatrix} 50 \\ 0 \\ 1 \end{bmatrix}$$

Fig. 2. Rotation Example – The result of rotating the point (0,0) in a 51 x 51 matrix 90^o counter clockwise around the center point – (25,25).

The Mutate operation is used to alter the form of an image. The mutation is accomplished by mapping an image into a form where every pixel is a 5-tuple, (x,y,r,g,b), where (x,y) is the coordinate location of the pixel and (r,g,b) is the color component. Each (x,y) component is put into a 3 x 1 vector (the last element of which is usually 1). This vector is then multiplied by a 3x3 matrix in order to get a desired effect. An example of a 90^o counter-clockwise rotation is in Figure 2. Using interpolation as needed, the resulting image is then mapped back to pixel format, with the order of the pixels defined by their location in the file.

The Modify operation is used to change colors within an image. The Modify operation accepts up to 6 parameters, which correspond to old and new values for the RGB color. A color can be omitted from the operation, and that implies a "don't care" condition for that color. Thus, Modify(Red(60,100)) would modify all pixels having a red value of 60 to a red value of 100, no matter what the values

of the green and blue components, while Modify(Red(60,100),Green(60,100), Blue(60,100)) would change only a pixel with 60/60/60 to 100/100/100. Ranges of values can be used for the old value of the pixels. Thus, an entire image can be turned black by Modify(Red(0...255,0),Green(0...255,0), Blue(0...255,0)).

The Combine operation is used to achieve certain effects which are based on localized portions of the image. For example, a blur effect is achieved by making each pixel look a little bit more like the pixels closest to it. Thus, the value of a pixel is computed by taking a weighted average of the pixel and the 8 pixels bordering it. See Figure 3 for an example of a simple blur matrix. Several other effects, such as distort and sharpen, are modeled as combinations of color values.

1	2	1
2	4	2
1	2	1

Weights of pixels

(50,50,50)	(25,50,100)	(50,50,50)
(25,50,100)	(200,200,200)	(25,50,100)
(50,50,50)	(25,50,100)	(50,50,50)

Example original pixels

Fig. 3. Combine Example – Sum of weights divided by 16 and rounded to the nearest integer results in a pixel value for the center of (75, 88, 112).

3 Implementation Issues

In Section 2, several system requirements are listed. In this section, we examine the implementation issues involved with the creation and maintenance of specifications and the upkeep of the transform-dags. Each of these issues is related to the natural uses of a MMDBMS. Instantiation results from a user requesting a transform object. The creation of specifications results from editing operations being applied to opened images. The maintenance of specifications and the upkeep of a transform-dag is a required to support deletion and modification of objects. It should be noted that naive instantiation of images is straightforward processing of the specifications, but *efficient* instantiation is beyond the scope of this paper.

3.1 Creation of Specifications

When an image is requested by an editor, a new object is created. This object is stored in the same *transform-of-***class** as the requested object, and is a child of the requested transform object. The child's long specification is the same as its parent's, but the child's short specification contains only a reference to the parent. Since the short specification defines the differences between a parent and its child, this short specification implies that the image associated with the child object is identical to the parent's image. When the image is returned to the editor, this child object is prepared to receive the edits performed by the user.

The reason for this is that the requested object is *permanent* in the database and may not be changed. This is needed to ensure that other transforms defined on this object will not be invalidated by updates to the permanent object. By contrast, the new object is a *transient* object and can be modified.

After the image is returned to the editor, specifications are added to the new transient object as users edit the image. The editor submits specification texts to the MMDBMS. For most operations, the MMDBMS creates a SE object with the text the same as submitted by the editor, the base being the oid corresponding to the object, and the offset being 0 (the only exception is the first operation which sets the offset to -1). However, merge operations are more complicated. For a merge operation, two SE's are added to the list. The first SE has the "source" of the merge for the base and an offset of -1. The text of the SE is null. The second SE has the destination image as its offset. This offset is the oid of the SE which was at the tail of the specification before the merge operation. The text has the merge command with the parameter being the location of the new information within the destination image. If the source object is transient, it is upgraded to stable to prevent deletion anomalies.

As an extended example, consider the case where images I_1 and I_2 have been edited by a user. In the database, there exists objects O_1 and O_2 which correspond to the results of applying editing operations to the respective base objects. Now assume that the user selects a part of I_2, copies it and pastes it into I_1. When the selection is performed on I_2, the *define* command along with the points defining the region are sent to the database. An SE object, S_1, is added to the short and long specifications of O_2. S_1 has text set to the define command with its parameters, the offset is set to zero, and the base[1] is set to a default value. However, the copy and paste creates a merge operation, which is more complicated.

With a merge operation, the database first changes the state of O_2 to stable. Next, a new object, O_3, is created as the child of O_2. Any further operations on I_2 will result in updates to the specification of O_3. For further information about the creation of O_3, see Section 3.2. A new SE object, S_2 is added to the short and long specification of O_1. O_2 is added to the parent set of O_1, and O_1 is added to the children of O_2. The text of S_2 is empty, but the base is the object id of O_2 and the offset is -1. A second SE object, S_3, is used to hold the operation. The command of S_3 is *merge*, and the parameter is the location in I_1 for the new information, the base[4] is set to the default value, and the offset is the oid of the SE which defines the destination image. In this case, that would be S_A. A chart of the short specifications is in Figure 4.

3.2 Insertion of New Images

Recall from Section 2 that we assume the editing tool can interface with the MMDBMS. This requires the tool to submit model language commands to the database system, and accept images from the database as described in

[1] The base is not used in instantiation when the offset is greater than or equal to zero

O_1				O_2				O_3			
oid:1				**oid:**2				**oid:**3			
SE	command	base	offset	SE	command	base	offset	SE	command	base	offset
S_A	NULL	$I_1.oid$	-1	S_B	NULL	$I_2.oid$	-1	S_C	NULL	2	-1
S_2	NULL	2	-1	S_1	define	0	0				
S_3	merge	0	*oid of* S_A								

Fig. 4. Merge with SE's

Section 3.1. However, the editing tool must also interact with the database in order to allow the system to maintain a correct specification of objects. Thus, in addition to the model language commands, the editor must also support additional commands related to storing and retrieving objects and transforms from a MMDBMS. These commands do cause an additional overhead, as the editor must submit them to the database system. However, this overhead is small compared to the effort which must be expended by a user in searching for images in an unstructured environment.

The first such command is **Open**. The purpose of an Open command is to allow the editor to have access to a particular media object. There are many levels of interaction which can exist between the editing tool and the database system with respect to this operation. For example, the database could require that the tool creates a query to select the object, or that the tool has a handle or pointer to the object, via browsing through the MMDBMS interface or as a direct association link from another object. Since we are building a simple prototype system, we assume the deepest level of interaction and require the editing tool to pass the actual object id to the database system. In actual practice, this would not be the best system since it would require too much information to be passed to the editor. A more general query or browsing based interface would be easier to implement across multiple MMDBMSs.

Upon receiving the Open command for object O, the database system determines if the class of O is transformable, or if it is a *transform-of*-**class**. If it is not, then the image is from a class which stores the binary representation of all images, and the image is simply returned to the editor and no further work is done. If the editor submits LML commands related to this object, the system ignores them. However, if the class is transformable or a *transform-of*-**class**, then the system creates a new object, O_1, which is the new transform of the current object, O. The object O_1 is stored as a new object in the *transform-of*-**class**. The parent of O_1 is O, and if O is a stored object, then it is the root of a new transform-dag. In the short specification, the first SE for O_1 has a null text field, an offset of -1 and the base is the oid of O. The long specification of O_1 is the same as the long specification of O. The image associated with O_1 is instantiated and returned to the user.

When the user is finished creating a new object, a second new command– **Save** is executed. Since the editing tool does not know if the class is transformable or not, the editor submits the entire object with the save command. If the class is not transformable, the new object is stored in the class, and the additional information, such as creation date and author, is added to the object. If the object is transformable, the object data is discarded, and the transform object, o_1, is now made permanent in the database. As with the non-transformable case, the additional data is also added. Clearly, if the editor is aware that the object is a transform, then significant network bandwidth can be saved by not transmitting the object. However, we are assuming the worst case to show how the database could handle the unwanted data.

If the user terminates the session without saving the object, then the editing tool should send an **Abort** message. When this message is received, the MMDBMS responds as a traditional database would to an abort of a transaction. Uncommitted data is removed from the system. However, in the case of transformable objects, it is certain that uncommitted data exists, and furthermore it must be treated as a deletion (see Section 3.3).

3.3 Deletions and Modifications

We use the principal of *transform independence* to resolve deletions and modifications of an image. View independence means that the user should not be aware if the object is a transform or a stored object. This means that deletions and modifications to an image should not alter any transforms defined on that image. In the case of deletion, the object is removed from the transform-dag. This requires that the children of the object now become children of the deleted object's parents. There are two cases which must be considered here. First, if the deleted image is stored (i.e., not a transform), then its children must now become stored objects since their specifications will be invalidated by the deletion. In order to do this, the children are instantiated as described in Section 3.1. They are then inserted as new objects into the database. The transform-dag is then followed from all of the newly instantiated objects. Each child in the dag must have its long specification rebuilt from the existing short specifications. To rebuild a long specification, the short specification is appended to the end of the long specifications of a transform's parents. The specifications must be re-optimized in order to improve instantiation speed, as any optimizations done could be destroyed in this process. Once this is complete, the transforms corresponding to the instantiated objects are deleted from the system.

The second case is when the deleted object is not stored, and thus has both parents and children. In this case, the long specifications are not changed, but the short specifications of the children must now include information from the deleted parent. Let the deleted parent be O_1, and the child be O. In the short specification of O, there exists an SE object, S_1, with base equal to the object id of O_1 and offset of -1. S_1 is removed from the specification, and the entire short specification of O_1 is put in its place. Furthermore, all references to the oid of S_1 are replaced by the oid of the last SE object in the newly added list.

Optimization of the short specification could be performed, but since it is not used to instantiate the object, it is not needed.

Note that if a transient object is not saved in the database, the same process must be followed. Thus, from the extended example in Section 3.1 and Figure 4, if O_2 is not saved in the database, then the short specification of O_2 is inserted in the short specification of O_1 in place of the SE object S_2.

Modifications of images are implemented by treating them as a delete followed by an insertion. Thus, if a stored object is modified, all of the children are instantiated as new stored objects, and the long specifications of all descendants in all transform-dags are updated. The modified object is stored back in the database. If a transform object is modified, it is removed from the transform-dag and the appropriate alterations are made to the short specifications of its former children. However, the modified object is not instantiated. Merely its specification is extended to include the modifications.

4 Conclusion and Future Work

Using transformations to represent images is a powerful, albeit complex, mechanism. A database system which uses transformations can save tremendous amounts of space. Also, transformation based representations do not suffer from image degradation from repeated compressing and decompressing of images. Thus, such a system provides better images and can store more images in the same amount of space.

However, the complexity of transformational representations carries a cost. The most obvious cost involves the time required to instantiate transformational representations. There is also the additional cost of database support for multimedia editing. Our paper presents a model that can be used to store the media objects using transformational representations. In doing so, we present the meta-structure, model language, and implementation techniques for object instantiation, insertion, deletion, and modification.

We propose the use of a *transform-dag* to represent the history of the creation of an image. An arc from object o_1 to o_2 means that o_1 was used to create o_2. The roots of the dag are the stored objects. Each object has two specifications which are the transformations needed to instantiate the image. One of these specifications is called the *short* specification, and the other is the *long* specification. The short specification indicates how the image would be reconstructed from its parents in the transform-dag. The long specification is used to instantiate the image from the stored objects or the roots of the transform-dag.

Two specifications are required to efficiently handle instantiations and deletions from the database. The long specification would be used to instantiate the image, thus reducing the amount of intermediate work that would be required. However, if only long specifications are used, and the stored object is deleted, then all of the derived objects would have to be instantiated as well. Thus, by allowing the transform-dag to have more than one level, it is possible to instantiate fewer objects on a deletion.

The transformations used here are based on a formal model of image editing derived from the Image Algebra in [10]. The system proposed here assumes that editors translate operations submitted by users into the transformational model, and then send them to the database. A simple LML consisting of five operations – Combine, Define, Merge, Modify and Mutate – is presented.

This work is part of an ongoing project to create a prototype system which can fully exploit transformational representations of images, audio and video. The control logic for the work presented in this paper is available via anonymous ftp from mercury.baylor.edu. The program uses a slightly different version of specification entry objects, but otherwise conforms to the algorithms presented here. In addition to this work, there are several other projects related to integrating transforms within a MMDBMS. First, the image LML must be completed. Issues such as completeness, minimality and independence are explored in [1], but alternative color models and quantization must be added to the model. Second, a tool for instantiating images from the image LML must be designed and implemented. This tool will also be used to instantiate transforms for user requests and when deletions require that the system instantiate images.

Also, it should be noted that the long specification of an object should not be the same as the pure concatenation of the short specifications of its ancestors. The long specifications should be optimized to improve instantiation time. For example, since size manipulations and rotations are accomplished through the mutate operation, which is matrix multiplication, a rotation and an increase in size can be combined into a single mutate operation in the specification. In order to do optimization, we must understand the compatibility of the operations, which is complicated by the large parameters taken by the commands.

This analysis of the LML commands for optimization also provides an opportunity for exploiting specifications for content-based retrieval. Content-based retrieval is finding an image based on the actual picture, rather than descriptions of it. There is much research in this complex area (see, e.g., [5,7,4,15]). However, the use of specifications provides semantic information not generally available. Our plan is to use a traditional content-based retrieval program on stored images, and then combine those results with the specifications to determine whether or not transforms satisfy the query. As a very simple example, if an image does not have a picture of a red balloon in it, a crop of that image will not either.

Finally, all of the work planned for images is also planned for audio and video. We will develop an LML for both audio and video. We will build editors which can submit operations to a system, and we will be able to instantiate the data from the specifications. We will also optimize the specifications. However, the work done in this paper will not be repeated, as the basic ideas presented here will hold in continuous media.

Thus, the object-oriented meta-structure for the transformational representations of images, is actually the meta-structure for all media types. The use of a dag to indicate the history of an object and a specification to define its form are the basic building blocks of a system which has tremendous capabilities in

terms of reducing the amount of space used by multimedia databases, and to improve the quality of the media at the same time.

References

1. L. Brown, L. Gruenwald, and G. Speegle. Testing a set of image processing operations for completeness. In *Proceedings of the 2nd Conference on Multimedia Information Systems*, April 1997. 127-134. 90, 94, 101
2. J. Cheng, N. Mattos, D. Chamberlin, and L. DeMiciel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994. 90, 90
3. A. Giles. *JPEGView Help*, 1994. Online help system. 90
4. Y. Gong. An image database system with content capturing and fast image indexing abilites. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems*, pages 121–130, May 1994. 101
5. V. Gudivada and V. Ragshavan, editors. *Computer : Finding the Right Image.* IEEE Computer Society, September 1995. 101
6. MPEG Integration Group. SO/IEC JTC1/SC29/WG11 coding of moving pictures and audio information, March 1996. N1195 MPEG96/ MPEG-4 SNHC Call For Proposals. ISO. 94
7. H. Jagadish. *Multimedia Database Systems: Issues and Research Directions*, chapter Indexing for Retrieval by Similarity, pages 165–184. Springer-Verlag, 1996. 101
8. S. Khoshafian and B. Baker. *Multimedia and Imaging Databases.* Morgan Kaufman, 1996. 89
9. W. Kim. *Introduction to Object-Oriented Databases.* MIT Press, 1990. 90
10. G. Ritter. Image algebra, 1995. Available via anonymous ftp from ftp.cis.ufl.edu in /pub/src/ia/documents. 94, 101
11. G. Speegle. Views as metadata in multimedia databases. *Proceedings of the ACM Multimedia '94 Conference Workshop on Multimedia Database Management Systems*, pages 19–26, October 1994. 90
12. G. Speegle. Views of media objects in multimedia databases. *Proceedings of the International Workshop on Multi-Media Database Management Systems*, pages 20–29, August 1995. 90
13. G. Schloss and M. Winblatt. Building temporal structures in a layered multimedia data model. *Proceedings of ACM Multimedia 94*, pages 271–278, October 1994. 90, 90
14. J. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1988. 90
15. A. Yoshitaka, S. Kishida, M. Hirakawa, and T. Ichikawa. Knowledge-assisted content-based retrieval for multimedia databases. *Proceedings of the International Conference on Multimedia Computing and Systems*, pages 131–139, May 1994. 101

Establishing a Knowledge Base to Assist Integration of Heterogeneous Databases

D. D. Karunaratna[1], W. A. Gray[2], and N. J. Fiddian[2]

[1] Department of Computer Science, University of Colombo, Sri Lanka
[2] Department of Computer Science, University of Wales-Cardiff, UK
{scmddk,wag,njf}@cs.cf.ac.uk

Abstract. In this paper the establishment of a supporting knowledge base for a loosely-coupled federation of heterogeneous databases available over a computer network is described. In such a federation, user information requirements are not static. The changes and evolution in user information needs reflect both the changing information available in the federation as databases join and leave it and their contents change, and the varied requirements the users have of the information available in the federation. As a user's awareness of the contents of the federation improves, their requirements from it often evolve. This means that users need different integrating views if they are to realise its full potential. The knowledge base will be used to help users of the federated databases to create views of the federation and to determine the available information in the federation.

The knowledge base (KB) is created and evolved incrementally by analysing both meta-data of databases as they join the federation and views as they are created over the databases in the federation. The KB is organised as a semantic network of concept clusters, and can be viewed as a conceptual model describing the semantics of all databases in the federation. It records not only concepts and their inter-relationships defined in schemas but also their frequency of occurrence within the federation, and is used both as a semantic dictionary and as a basis to enrich schemas semantically during the detection of semantic relationships among schema components when creating user views. We semantically enrich schemas prior to integration by generating the best sub-network that spans the concepts common to the schema and the knowledge base and then by unifying this sub-network with the schema. Hence the terms in schemas are given interpretations not by considering them in isolation but by taking into account contexts, derived from the knowledge base with respect to those schema terms.

An architecture to facilitate the establishment of this knowledge base and to exploit its functionality is described. The novelty of our research is that we create a knowledge base for a federation by gathering and organising information from it to assist groups of users to create dynamic and flexible views over the databases in the federation which meet their dynamic information requirements. This knowledge base enables re-use of the semantics when creating views.

Keywords: loosely-coupled federation, heterogeneous databases, semantic dictionary, database discovery.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 103–118, 1998.

1 Introduction

Recent progress in network and database technologies has changed data processing requirements and capabilities dramatically. Users and application programs are increasingly requiring access to data in an integrated form, from multiple pre-existing databas es [6,34,42], which are typically autonomous and located on heterogeneous software and hardware platforms. For many of these databases, design documentation, high level representation of domain models (e.g. ER, OMT representations etc.) and additional domain knowledge from the DB designers may not be available. Different users have different needs for integrating data and their requirements may change over time and as new databases become available. This means that the same database may participate in many different ways in multiple integration efforts, hence different views of the databases need to be constructed.

A system that supports operations on multiple databases (possibly autonomous and heterogeneous) is usually referred to as a *multidatabase system* [3,22,23,32]. There are two popular system architectures for multidatab ases: tightly-coupled federation and loosely-coupled federation [8,20,39]. In a tightly-coupled federation, data is accessed using a global schema(s) created and managed by a federated database administrator(s). In a loosely-coupled fed eration it is the user's responsibility to create and maintain the federation's integration regime. They may access data either by means of views defined over multiple databases [18,19] or by defining a query using a multidatabase language ţteLitw88,Litw90 which enables users to link concepts within queries.

With the growth and widespread popularity of the Internet, the number of databases available for public access is ever-increasing both in number and size, while at the same time the advances in remote resource access interface technologies (e.g. JDBC, OD BC) offer the possibility of accessing these databases from around the world across heterogeneous hardware and software platforms. It is envisaged that the primary issue in the future will not be how to efficiently process data that is known to be relevan t, but rather to determine which data is relevant for a given user requirement or an application and to link relevant information together in a changing and evolving situation[27,35,36]. The creation of an environment that permits the con trolled sharing and exchange of information among multiple heterogeneous databases is also identified as a key issue in future database research [17,40]. In this environment we believe that the loosely-coupled architecture provides a more via ble solution to global users' evolving and changing information requirements across the disparate databases available over a network, as it naturally supports multiple views which are dynamic, and it also avoids the need to create a global integrated sche ma.

In our research the main focus is on loosely-coupled federations where views are created and used to define a user's requirements to access and link together data from multiple databases. In a loosely-coupled federation, databases may join and leave as th ey please. In addition, global users may create, modify and enhance views in many different ways to meet their changing information requirements in a flexible manner. Some of the major issues to be addressed when

creating multidatabase views in such an en vironment are how to locate appropriate databases and data for some application and how to determine objects in different databases that are semantically related, and then resolve the schematic differences among these objects [2,21,29]. This must be achieved in a way that makes it possible for the federation to evolve, and easy for a user to create new views and change existing views. A wealth of research has been done on database integration but most of this work assumes that the releva nt databases and data can be identified by some means during a pre-integration phase [1] and that integration is a static outcome which is comprehensive and will be relevant to all future applications. Furthermore, much published research in thi s area concentrates either on a one-shot integration approach which creates a single global schema or an integration via a multidatabase query [34]. Significantly, little or no emphasis has been placed on the re-usability of knowledge elicited d uring the schema integration process, to create different views which dynamically reflect a user's changing information requirements.

In this paper we investigate how to establish a knowledge base (KB) to assist users to create views which integrate schemas in a loosely-coupled federation of heterogeneous databases in a dynamic and flexible manner. In our approach we assume that knowled ge about a database is elicited at the time it joins the federation, but is not necessarily used at that time to create an integrated schema. The knowledge base we establish evolves as new databases join the federation. It will also evolve as new views ar e created by users as it will capture the semantics used in creating these views. The knowledge gathered in each step is utilised in subsequent integration efforts to improve the new views created by users. The KB we propose is not a global schema. Its in tention is not to create a view(s) specifically oriented towards a particular user group(s) but rather to capture information which can be used to support an environment in which users can create multiple, dynamic and flexible integrated views with less e ffort. Thus the knowledge base cannot be used directly as an integrated schema which helps users access data from the component databases in a federation.

We use a bottom-up strategy to build the knowledge base. With the help of DB owners, we initially use the schemas of the DBs in the federation to create a set of acyclic graphs of terms (referred to as *schema structures*). These schema structures are then merged successively with the assistance of the database owners to build the federated knowledge base. At this stage we are attempting to capture the knowledge about the databases known to their owners. Finally, the KB is utilised as a semantic dicti onary to assist users in generating integrated views to access data from multiple databases in the federation when they are creating applications based on new data requirements and to find out about the information held in the federation.

For our approach, it is not a pre-requisite for the schemas of the component databases to be available in some conceptual data model such as ER or OMT. Neither do we expect to link schema components with an existing global knowledge base such as a thesaur us, WordNet [26], some form of concept hierarchies, ontologies [15,16], etc, to assign interpretations as in [2,14,25,43]. These

approaches to providing a knowledge base assume that an available thesaurus wi ll adequately cover all concepts in the federated system. Whereas we build our thesaurus from the concepts in the federation thereby ensuring its relevance to the contents of the federation.

The remainder of this paper is organised as follows: in section 2 we give a brief review of related work. An architecture to support the establishment of our knowledge base and to exploit its functionality is described in section 3. Section 4 explains th e structure of the knowledge base and how it is created and modified. In section 5 we explain the functionality that the knowledge base can provide to assist users in building dynamic and flexible views. Finally, section 6 draws conclusions.

2 Related Research

In early integration methodologies [28,30], identification of relationships between different database schema objects was considered to be the responsibility of database integrators. These integrators usually reasoned about the meaning and po ssible semantic relationships among schema elements in terms of their attributes and their properties (e.g. names, domains, integrity constraints, security constraints, etc). A measure based on the number of common attributes is normally used, as a basis to determine the resemblance of schema elements. The discovered relationships are then represented as assertions to be used by the schema integration tools [7,33,38,41] to integrate schemas (possibly automatically). With the realis ation that the manual identification of semantic relationships among schema objects is difficult, complex and time consuming and that the process cannot be fully automated [37], research began to explore the possibility of developing tools to ass ist users in this task.

The algorithms used by such tools to determine semantic relationships are often heuristic in nature. The fundamental operation during the identification process is comparison but the type and depth of information used for comparisons vary from tool to too l. Many of the early tools [10,41] used techniques based on the early manual approaches and relied heavily on the information available in the schemas. They basically compared the syntactic properties of attributes in pairs to determine their equivalence, and used attribute equivalence as a heuristic to determine the semantic similarities among schema objects (e.g. relations, classes, etc). No attempt was made in this process to reuse the knowledge elicited from integrating one group of schem as in any subsequent integrations of other (related) schema groups. With the realization that schema information contents alone is not sufficient to determine the semantic similarity between schema objects [11,21,2], attempts were made t o enrich the semantics of the schema elements prior to the integration process. This semantic enrichment is done basically in two ways: by eliciting knowledge from the database owner and building a semantic model over the schema, or by linking schema elem ents with existing knowledge bases (KBs). Tools that attempt to create conceptual models (e.g. ER, OMT) on top of schemas prior to integration can be consid-

ered to belong to the first category, while schema integration tools that utilise already existing ontologies, terminological networks such as WordNet, concept hierarchies, etc. fit into the second category.

Tools and techniques to support the establishment of KBs for database schema integration are reported in [2,4,9]. Many of the KBs proposed [2,4] are strict concept hierarchies where the generalisation/specialisation rela tionship is used to organise concepts. Other types of relationship such as positive-association and aggregation are included in a number of projects [13,43], but how these types of relationship are incorporated among the concepts in the KB i s not mentioned. To the best of our knowledge the inference of semantic relationships among schema components (or DBs) by pooling the knowledge about several databases into a common repository which is then used as the KB to support the integration proces s is novel.

3 System Architecture

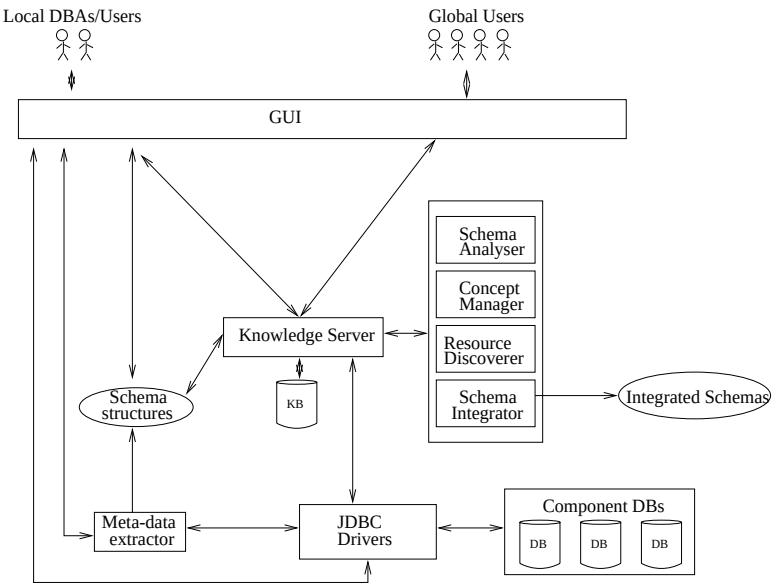


Fig. 1. The Architecture of our System

The architecture we adopt is depicted in Figure 1. Its major components are the Meta-Data Extractor (MDE), the Knowledge Server (KS) and the Graphical User Interface (GUI). The MDE interacts with the component databases (DBs) in the fe deration via JDBC drivers to extract their intensional data, and builds an associative network based model (*schema structure*) locally for each

participating DB in the federation. These schema structures are then analysed to discover implicit relatio nships among different schema elements, enriched with semantics elicited from the DB owners, and merged with the KB. The KB is maintained by the KS which provides the functionality necessary for its establishment and evolution and for its role in assistin g users in their integration efforts.

The GUI component is the primary interface between the users, the knowledge server and the component databases. It allows the DB owners to view the schema structures created by the MDE as acyclic graphs of terms and provides functionality to modify schema structures and enhance the semantics of the elements in them. In addition the global users may use the GUI to invoke functionality provided by the KS.

3.1 Schema structures

```
catalogued_book(isbn,author,title,publisher,year,classmark)
reservation(isbn,name,address,date_reserved).
book_loan(book_no,isbn,shelf,name,address,date_due_back)
person(name,address,status)
loan_status(status,loan_period,max_num_books).
```

Fig. 2. An example of a specimen relational schema (based on a schema from [31])

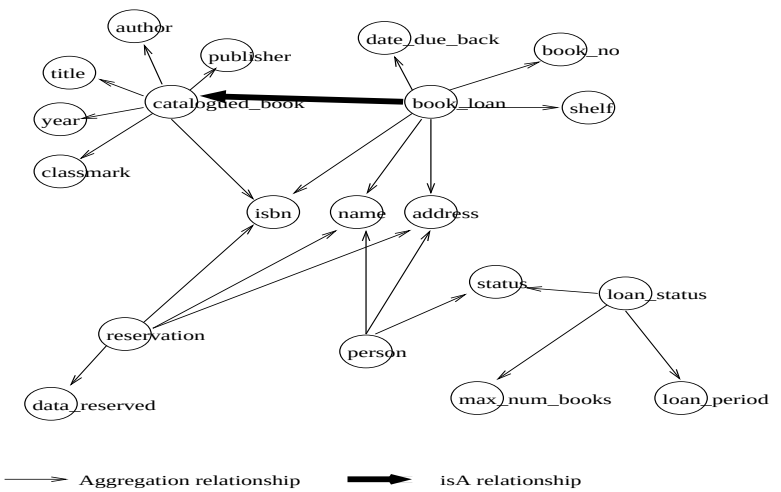


Fig. 3. An example of a Schema structure

The primary information source used to build (and evolve) the KB is the meta-data of the component databases in a federation. The DBs in the federation may be implemented in different database management systems (DBMSs) with different data models (e.g. re lational, hierarchical, network, object oriented, etc.). This type of heterogeneity in a multidatabase system is referred to as *syntactic heterogeneity* [5,3]. The solution commonly adopted to overcome syntactic heterogeneity is to use a canonical data model and to map all schemas in a federation to this canonical model. We have determined that the meta-data required to build the KB can be provided by building associative networks over each database in the federation. Hence in our appr oach the canonical data model used is based on associative networks, and the meta-data models generated on top of the component DBs by using this data model are referred to as *schema structures.*

3.2 Canonical data model

We organise the meta-data of a schema as an associative network of terms, where terms represent either schema objects (e.g. relation, entity, class, etc.) or attributes of schema objects defined in the corresponding DB schema. Terms are related with other terms via binary relationships (*links*). The terms and links have *properties* some of which are essential. Each property is given a *label* (e.g. name, strength, etc.) and has a *value(s).* The labels classify properties into *types* . In the following sections we use labels to identify properties. Two essential properties of a link are its *strength* and its *link-type.* The strength of a link represents how closely the connected terms are related in the DB and takes a value in the interval $(0, 1]$. A link with strength 1 indicates a definite relationship. All the other values represent tentative associations. The link property *link-type* can take a value from the set {aggregation, isA, positive-association, synonym}.

The links between terms in a schema structure are generated as follows :

aggregation : links two terms T_1 and T_2 when T_1 corresponds to a schema object O_1 and T_2 corresponds to an attribute A_i of the object O_1.

isA : links terms T_1 and T_2 when T_1 and T_2 denote two schema objects O_1 and O_2, respectively, and a specialisation/generalisation relationship exists between them.

positive-association : links terms when an association other than *aggregation* and *isA* is defined between objects in the schema.

synonym : links two differently named schema terms when they correspond to two schema object attributes that denote the same real-world aspect. For example in relational DBs, the same attribute may exist with different names in different relat ions to create explicit relationships between relations.

Since *aggregation*, *positive-association* and *isA* links are used to link terms only when such associations are explicitly defined or can be identified by the DB

owners, their strengths are always assigned the value 1. The tentative associa tions may be generated among terms by using schema analysis tools to guide DB owners to discover implicit relationships among schema elements. Once such relationships are discovered, DB owners may transform tentative links to definite links by changing th eir strength to 1. The implementation characteristics of schema objects and their attributes (e.g. data type, scale, precision) may be coded in schema structures as properties of their corresponding terms.

In Figure 2 a specimen relational schema is shown and its corresponding generated schema structure is shown in Figure 3. We expect the DB owners to identify different schema object attributes in their schema structures which denote the sa me real-world aspects and to merge such attributes as a single term (e.g. isbn in Figure 3) or to relate them with *synonym* links before merging the schema structures with the KB.

4 Structure, Creation and Modification of the KB

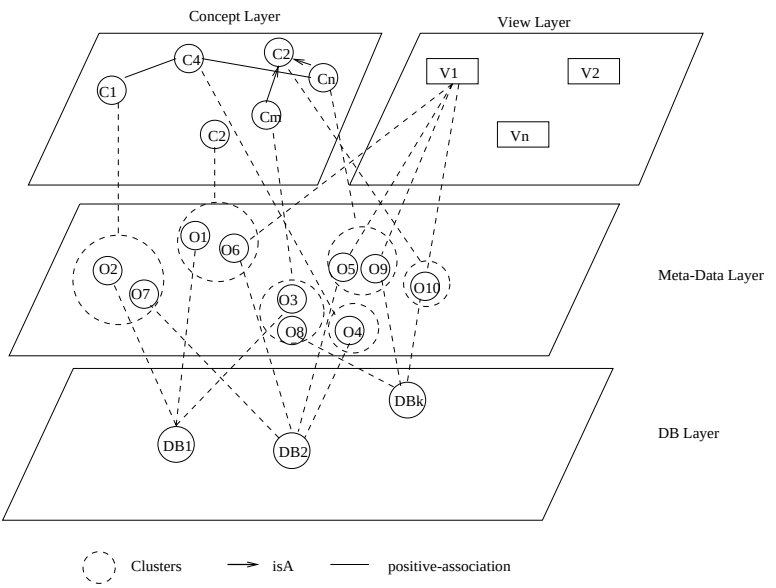


Fig. 4. The Structure of the KB

The KB can be viewed as composed of four layers: a *concept layer*, a *view layer*, a *meta-data layer* and a *DB layer*, as shown in Figure 4. The top left layer comprises concepts, each of which represents an object cluster in the meta-data layer. A concept in the top left layer can be thought of as a unit of knowledge representation that guides the process of classifying schema objects into clusters,

and a cluster can be considered as a logical grouping of schema objects. Concepts may be related with each other via binary relations (*links*). As in schema structures, links have properties. In addition to *strength* and *link-type* properties, they also have an essential property named *frequency*, which specifies how many times the corresponding link occurred in the federation and is being used in user views. In the concept layer, link strengths represent how closely concepts are related in the federation. The link property *link-type* here takes values from the set {aggregation, isA, positive-association}.

In the concept layer, *positive-associations* represent possible associations among concepts, and are generated as described in the following sections. Concepts possess attributes providing descriptive information about themselves. The attributes are connected with their corresponding concepts through aggregation links. When an attribute of a concept is further described by using a set of attributes, it is treated as another concept. Hence a concept may have attributes which are also concepts. Both concepts and attributes may possess properties. Two essential properties of concepts and attributes are *name* and *frequency*. A concept is allowed to have multiple names in the concept layer. The frequency of a concept (or a concept attribute) specifies in how many databases in a federation the concept (or the attribute) is defined.

The view layer stores information relevant to views generated by users over the federation. Its main function is to gather knowledge required to determine and notify view users, DB meta data changes that potentially affect their views.

The meta-data layer maintains all the schema structures submitted to the KS, by organising all terms that represent objects in schema structures into clusters (*object clusters*). All corresponding objects in a cluster are hypothesised to be semantica lly similar. This layer is intended to be used to assist users in reconciling schematic differences among semantically similar items during schema integration.

The DB layer maintains information about the databases whose schema objects are represented in the meta-data layer. It is used to link to the extensional data.

4.1 Merging schema structures with the KB

The KB is created by merging, in ladder fashion [1], the schema structures of the component DBs. Merging a schema structure with the KB changes all four layers of the KB. At the DB layer a new node is created to capture the information of the DB being merged. All meta-data in the schema structure is added to the meta-data layer and the terms in the schema structure are merged with concepts in the concept layer as described below.

4.2 Merging schema structures with the concept layer

During this process some of the terms in schema structures are merged with concepts in the concept layer while others are merged with attributes of existing

concepts. The approach we use to merge a schema structure with the KB comprises five stages: *common term set generation, sub-net(s) generation, sub-net(s) and schema structure unification, merge terms with concepts*, and *merge links*, which are carried out once in that order to merge a schema structure with the KB. The first three stages of this process are aimed at establishing mappings between schema structure terms and concepts by considering all schema terms as a single unit of related terms. The last two stages assimilate schema structure knowledge into the KB.

In the *common term set generation* stage, a set S of terms common to both the schema structure and the KB is generated. The common terms may exist in the KB either as concepts or as attributes of concepts.

During the *sub-net(s) generation* stage, terms in the set S are spanned along aggregation and positive-association links in the concept layer whose strengths exceed a given threshold value, to obtain a sub-net (or a set of disjoint sub-nets) of conce pts that span all terms in S. We use the following algorithm to generate the sub-net(s) that span all the common terms in the set S. Initially a term t_0 in the set S is selected as a seed term, and all concepts in the concept layer with the same name are taken as starting concepts. All starting concepts are then spanned to a given depth (d), generating a number of distinct sub-net(s) of concepts. We define the span depth as the number of links traversed by the spanning process from the starting concep t. Out of these sub-nets, the sub-net having the maximum number of terms in the set S is selected as the current sub-net. In the next step, a new set S is created by removing the terms that are in the current sub-net from the initial set S. One of the ter ms in this new set is selected as the new seed and its corresponding concepts are spanned as above, until either one of the concepts generated during the spanning process matches a concept in the current sub-net or all concepts span to the maximum depth, d, without matching a concept in the current sub-net. In the former case, the sub-net generated by the spanning process is merged with the current sub-net to generate a new current sub-net to be used in the next iteration. In the latter case, the sub-net(s) that has the maximum number of terms in the set S is selected as the best span and is added to the current sub-net. Then a new set is created by removing the terms in S that appear in the best span, and the process is repeated till the set S is empty.

In the *sub-net(s) and schema structure unification* stage, concepts in the spanned sub-net(s) are unified with terms in concept structures that represent schema objects to determine the most suitable concept for each term to merge with. The final res ult of this stage is a set of ordered pairs (T_i,{C_1,C_2,C_3...C_n}), where T_i is a term in the schema structure representing an object and the C_is represent a suitable concept for T_i to merge with. With the assistance of the DB owne rs, the above generated ordered pairs are converted into a set of ordered pairs of the form (T_i,C_i), where C_i is the concept with which the term T_i is to be merged in the next stage. Then the T_is in the meta-data layer are moved into th e clusters represented by the concept C_is. When several concepts match closely to a given schema object term, then all such matching concepts that are not linked

via a common parent are also unified to determine their inter-relationships and the result s are shown to the DB owner. He may inspect these results to discover scattered descriptions of the same concepts. If such concepts are found, they may be merged together.

In the *merge terms with concepts* stage, terms in the schema structure are merged with the identified concepts in the concept layer and the properties (e.g. frequency) of the concepts and their attributes are updated appropriately. For the schema obj ect terms where no suitable terms are found in the concept layer, new concepts are created.

During the *merge link stage*, new links may be created among concepts, and the existing links properties are updated. We create a positive-association link (or update their frequencies) among two concepts C_i and C_j when either a positive associ ation is defined between T_i and T_j in the schema structure or there is a path over the aggregation and positive-association links whose length is less than a given value between T_i and T_j in the schema structure. Based on the frequency values of concepts and links, a number of different computations can be devised to calculate link strengths. After experimenting with a number of different computations in our prototype, we chose to use the following equations to compute link strengths, by takin g into account the similarity measures established in cluster analysis [12]. We have observed that the selected computations yield a fewer number of more focused disjoint sub-nets during sub-net generation stage.

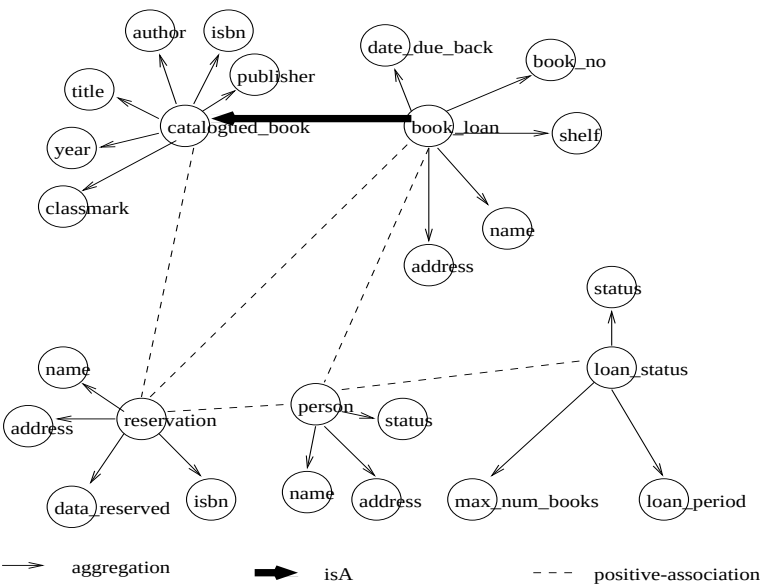


Fig. 5. Structure of KB concept layer after merging the schema structure shown in figure 3 with an empty KB

- aggregation link strength connecting concept p with its attribute i = F^a_{pi}/F^c_p
- positive-association link strength connecting the concept i with the concept j = $F^l_{ij}/\min(F^c_i,F^c_j)$
- isA link strength = 1

where F^c_i is the frequency of concept i, F^c_{pi} is the frequency of attribute i of concept p, F^l_{ij} is the frequency of the link connecting concepts i and j.

The resulting conceptual layer of the KB is shown in Figure 5, after the schema structure shown in Figure 3 is merged with an initial empty KB.

5 Functionality of the KB

In the KB, schema objects considered to be semantically similar are grouped into clusters and a concept is attached at a higher level to represent each cluster. In addition, links are maintained between schema objects and databases from which they came. H ence this knowledge organisation allows users to start discovering databases relevant for some application (DB discovery) from two levels, from the concept layer or from the DB layer. A user can initiate the DB discovery process from the concept layer by specifying a set of concepts $\{C_1,C_2,C_3...C_n\}$ that he thinks might be relevant for his application. These concepts, C_is, lead to a set of clusters, and then to a collection of schema objects $\{O_1,O_2,O_3...O_m\}$ at the meta-data l ayer. By moving down from these schema objects, O_is, along the links maintained between the meta-data layer and DB layer, a set of databases $\{DB_1,DB_2,...DB_k\}$ in which the schema objects are defined can be determined. The set $\{DB_1,DB_2, ...DB_k\}$ is then sorted on a weight computed by considering the concepts to which these DB_is, relate and are presented to the user. Users may alternatively start the DB discovery process from the DB layer by selecting a candidate DB(s) from the fede ration. In this case the selected DB(s) is used to generate a set of concepts at the concept layer by moving along the links between the DB layer and the meta-data layer, followed by the links between the meta-data layer and the concept layer. The generat ed set of concepts are then used as before to discover DBs in the federation that define relevant schema objects.

Once a set of DBs relevant for some application is chosen, users may determine the relationships among the schema objects in them by determining their clusters, the concepts representing those clusters, and the associations existing among clusters at the concept layer. The schema objects belonging to the same cluster are considered to represent semantically similar real world entities, while the semantic relationship of schema objects belonging to different clusters can be determined by considering the re lationships between the concepts represented by these clusters.

When views are created over DBs in the federation, users may integrate schema objects in many different ways. The existing links used and the new links generated during these integration efforts are also recorded at the concept layer and this information is utilised to assist users in subsequent integrations.

The links maintained between each layer facilitate the easy modification of information in the three layers of the KB, during DB schema modification. When a DB leaves the federation the KB objects corresponding to it at the DB layer and meta-data layer ar e removed and the frequencies of relevant concepts and links at the concept layer are updated to reflect this change.

6 Conclusion and future work

We have described how a KB can be established over a loosely-coupled federation of heterogeneous DBs to assist users to integrate schemas. Our research is guided by the assumptions that:

- The schemas developed to represent similar real world domains typically share a set of common terms as schema object names and their attribute names.
- Semantic relationships among schema elements in a federation can be identified better by pooling meta-data of component DBs.

We do not depend on any pre-existing global taxonomies/dictionaries in building the KB. The KB we establish is dynamic and evolves with the federation. It is more germane to the federation as it uses mainly the meta information of the DBs from the schemas and the DB owners, and more comprehensive as it stores information necessary for a wide variety of activities such as information browsing, planning/preparing for an integration, resolving schematic conflicts, accessing data, etc.

An approach based on generating a sub-net spanning a set of terms, followed by unification, is used to merge DB meta-data with the KB. How the knowledge generated by pooling meta-data of component DBs is used to determine the strength of associations of c oncepts within the federation, and how these strengths are utilised to enhance the semantics of schema elements has been explained. Finally we described how the organisation of the KB helps users in database discovery and in determining semantic relations hips among schema objects.

The architecture and algorithms we have presented in this paper are based on a prototype system built to support users creating views over a set of library databases. This prototype was built to demonstrate that our knowledge base could be established and utilised to create views. Two library databases have been linked in the federation and a number of views have been created utilising the resulting knowledge base to assist in view creation. We intend to extend our research by building a work-bench of too ls to assist users to establish the KB and to utilise its functionality in their integration efforts. As our KB stores information at the DB layer to retrieve data from the component databases in the federation, we are also aiming to develop a query proce ssor to materialise user views created over the federation. In addition we intend to explore the possibility of using the global KB such as WordNet, ontologies, etc. to enrich and verify our KB.

References

1. C. Batini, M. Lenzerini, and S. B. Navathe. A comparative analysis of methodologies for database schema integration. *ACM Computing Surveys*, 18(4):323–364, December 1986. 105, 111
2. M. W. Bright and A. R. Hurson. Linguistic support for semantic identification and interpretation in multidatabases. In *Proceedings of the First International Workshop on Interoperability in Multidatabases*, pages 306–313, Los Alamitos, California, 1991. 105, 105, 106, 107, 107
3. M. W. Bright, A. R. Hurson, and S. H. Pakzad. A taxonomy and current issues in multidatabase systems. *IEEE Computer*, 25(3), March 1992. 104, 109
4. S. Castano and V. De Antonellis. Semantic dictionary design for database interoperability. In *Proceedings of the 13th International Conference on Data Engineering*, pages 43–54, Birmingham, U.K., April 1997. 107, 107
5. M. Castellanos and F. Saltor. Semantic enrichment of database schemas: An object oriented approach. In Y. Kabayashi, M. Rusinkiewiccz, and A. Sheth, editors, *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 71–78, Kyoto, Japan, 1991. 109
6. A. Chatterjee and A. Segev. Data manipulation in heterogeneous databases. *SIGMOD RECORD*, 20(4), December 1991. 104
7. S. Chawathe, H. G. Molina, J. Hammer, K. Ireland, Y. Papakonstantinous, J. Ullman, and J. Widom. The TSIMMIS project: Integration of heterogeneous information sources. In *Proceedings of the 10th IPSJ Conference*, pages 7–18, Tokyo, Japan, October 1994. 106
8. C. Collet, M. N. Huhns, and W. Shen. Resource integration using a large knowledge base in Carnot. *IEEE Computer*, pages 55–62, December 1991. 104
9. S. Dao, D. M. Keirsey, R. Williamson, S. Goldman, and C. P. Dolan. Smart data dictionary: A knowledge-object-oriented approach for interoperability of heterogeneous information management systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 88–91, Kyoto, Japan, 1991. 107
10. U. Dayal and H. Hwang. View definition and generalization for database integration in a multidatabase system. *IEEE Transactions on Software Engineering*, SE-10(6):628–645, November 1984. 106
11. B. EagleStone and N. Masood. Schema interpretation: An aid to schema analysis in federated database design. In *Engineering Federated Database Systems (EFDBS97), Proceedings of the International CAiSE97 Workshop*, University of Magdeberg, Germany, June 1997. 106
12. B. S. Everitt. *Cluster Analysis*. Edward Arbold, 3rd edition, 1993. 113
13. P. Fankhauser and E.J. Neuhold. Knowledge-based integration of heterogeneous databases. In *Proceedings of the IFIP WG 2.6 Conference on Semantics of Interoperable Database Systems(DS-5)*, pages 155–175, Lorne, Victoria, Australia, November 1992. 107
14. A. Goni, E. Mena, and I. Illarramendi. Querying heterogeneous and distributed data repositories using ontologies. In *Proceedings of the 7th European-Japanese Conference on Information Modelling and Knowledge Bases (IMKB'97)*, Toulouse, France, May 1997. 105
15. T. R. Gruber. A translation approach to portable ontology specifications. *Knowledge Acquisition*, 5(2):199–220, 1993. 105

16. T.R. Gruber. Toward principles for the design of ontologies used for knowledge sharing. Technical Report KSL 93-04, Knowledge Systems Laboratory, Stanford University, 1993. 105
17. J. Hammer and D. McLeod. An approach to resolving semantic heterogeneity in a federation of autonomous, heterogeneous database systems. *International Journal of Intelligent & Cooperative Information Systems*, 2(1):51–83, 1993. 104
18. D. Heimbigner and D. McLeod. Federated information bases - a preliminary report. In *Infotech State of the Art Report, Databases*, pages 223–232. Pergamon Infotech Limited, Maidenhead, Berkshire, England, 1981. 104
19. D. Heimbigner and D. McLeod. A federated architecture for information management. *ACM Transactions on Office Information Systems*, 3(3):253–278, July 1985. 104
20. V. Kashyap and A. Sheth. Schema correspondences between objects with semantic proximity. Technical Report DCS-TR-301, Department of Computer Science, Rutgers University, October 1993. 104
21. V. Kashyap and A. Sheth. Semantic and schematic similarities between database objects: a context-based approach. *VLDB Journal*, 5:276–304, 1996. 105, 106
22. W. Kim and J. Seo. Classifying schematic and data heterogeneity in multidatabase systems. *IEEE Computer*, pages 12–18, December 1991. 104
23. W. Litwin. From database systems to multidatabase systems: Why and how. In *Proceedings of the 6th British National Conference on Databases(BNCOD6)*, pages 161–188, Cardiff, U.K., 1988. 104
24. W. Litwin, L. Mark, and N. Roussopoulos. Interoperability of multiple autonomous databases. *ACM Computing Surveys*, 22(3):267–293, September 1990.
25. E. Mena, V. Kashyap, A. Sheth, and A. Illarramendi. OBSERVER: An approach for query processing in global information systems based on interoperation across pre-existing ontologies. In *Proceedings of the First IFCIS International Conference on Cooperative Information Systems(CoopIS'96)*, Brussels, Belgium, June 1996. 105
26. G. Miller. WordNet: A lexical database for English. *Communications of the ACM*, 38(11), November 1995. 105
27. S. Milliner, A. Bouguettaya, and M. Papazoglou. A scalable architecture for autonomous heterogeneous database interactions. In *Proceedings of the 21st VLDB Conference*, pages 515–526, Zurich, Switzerland, 1995. 104
28. A. Motro and P. Buneman. Constructing superviews. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 56–64, 1981. 106
29. S. Navathe and A. Savasere. A practical schema integration facility using an object-oriented data model. In O. A. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems, A Solution for Advanced Applications*. Prentice Hall, 1996. 105
30. S. B. Navathe and S. G. Gadgil. A methodology for view integration in logical database design. In *Proceedings of the 8th VLDB Conference*, pages 142–164, Mexico City, Mexico, September 1982. 106
31. Elizabeth Oxborrow. *Databases and Database Systems, Concepts and Issues*. Chartwell-Bratt(Publishing and Training Ltd), Sweden, 2nd edition, 1989. 108
32. E. Pitoura, O. Bukhres, and A. K. Elmagarmid. Object-oriented multidatabase systems: An overview. In O. Bukhres and A. K. Elmagarmid, editors, *Object-Oriented Multidatabase Systems, A Solution for Advanced Applications*. Prentice Hall, 1996. 104

33. M. A. Qutaishat. *A Schema Meta Integration System for a Logically Heterogeneous Object-Oriented Database Environment.* PhD thesis, Department of Computing Mathematics, University of Wales College of Cardiff, Cardiff, 1993. 106
34. A. Rosenthal and L. J. Seligman. Data integration in the large: The challenge of reuse. In *Proceedings of the 20th VLDB Conference*, Santiago, Chile, September 1994. 104, 105
35. A. Sheth and V. Kashyap. So far (Schematically) yet so near (Semantically). In *Proceedings of TC2-IFIP WG 2.6 Conference on Semantics of Interoperable Database Systems (DS-5)*, pages 283–312, Lorne, Victoria, Australia, November 1992. 104
36. A. P. Sheth. Semantic issues in multidatabase systems. *SIGMOD RECORD*, 20(4):5–9, December 1991. 104
37. A. P. Sheth, S. K. Gala, and S. B. Navathe. On automatic reasoning for schema integration. *International Journal of Intelligent and Co-operative Information Systems*, 2(1):23–50, March 1993. 106
38. A. P. Sheth, J. Larson, A. Cornelio, and S. B. Navathe. A tool for integrating conceptual schemas and user views. In *Proceedings of the 4th International Conference on Data Engineering*, pages 176–182, Los Angeles, CA, February 1988. 106
39. A. P. Sheth and J. P. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Computing Surveys*, 22(3):183–236, September 1990. 104
40. A. Silberschatz, M. Stonebraker, and J. D. Ullman. Database systems: Achievements and opportunities. *SIGMOD RECORD*, 19(4):23–31, December 1990. 104
41. J. De. Souza. SIS: A schema integration system. In *Proceedings of the 5th British National Conference on Databases (BNCOD5)*, pages 167–185, Canterbury, U.K., July 1986. 106, 106
42. S. Spaccapietra, C. Parent, and Y. Dupont. Model independent assertions for integration of heterogeneous schemas. *VLDB Journal*, 1:81–126, July 1992. 104
43. C. Yu, W. Sun, S. Dao, and D. Keirsey. Determining relationships among attributes for interoperability of multi-database systems. In *Proceedings of the First International Workshop on Interoperability in Multidatabase Systems*, pages 251–257, Kyoto, Japan, April 1991. 105, 107

Considering Integrity Constraints During Federated Database Design *

Stefan Conrad, Ingo Schmitt, and Can Türker

Otto-von-Guericke-Universität Magdeburg
Institut für Technische und Betriebliche Informationssysteme
Postfach 4120, D–39016 Magdeburg, Germany

{conrad,schmitt,tuerker}@iti.cs.uni-magdeburg.de

Abstract. Correct transformations and integrations of schemata within the process of federated database design have to encompass existing local integrity constraints. Most of the proposed methods for schema transformation and integration do not sufficiently consider explicit integrity constraints. In this paper we present an approach to deal with integrity constraints. Our approach bases on the idea to relate integrity constraints to extensions. A set of elementary operations for schema restructuring and integration is identified. For these operations we define major rules for dealing with integrity constraints. By means of a small example we then demonstrate the application of these rules.

1 Introduction

In many large organizations, different legacy data management systems are maintained and operated. These data management systems and the data managed by them have developed independently from each other. Among these systems there are database management systems but also merely file-based systems differing in several aspects such as data model, query language, and system architecture as well as the structure and semantics of the data managed. In this context the term 'heterogeneity of the data management systems' is commonly used. As a rule, applications for specific needs continue to be based on such systems. More recent applications often require access to the distributed databases but their implementation fails due to heterogeneity. *Federated database systems* [SL90] are designed to overcome this problem by integrating data of legacy database systems only virtually. That is, 'only' the local schemata are integrated to a federated schema. Global applications can retrieve and update federated data through the federated schema.

The main task during the design of a federated database is the transformation and integration of existing local schemata into a federated schema. The transformation [EJ95] and integration [LNE89,SPD92,RPG95,SS96a] of local schemata into a federated schema must fulfill the following requirements:

1. For each query against a local schema there exists a global query against the federated schema returning the same query result. This requirement avoids loss of information during the federation.

* This work was partly supported by the German Federal State Sachsen-Anhalt under grant number FKZ 1987A/0025 and 1987/2527R.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 119–133, 1998.

2. Each global update or insert operation can be propagated to the local DAMS correctly. That is, the local schemata must not be more restrictive than the federated schema. Therefore, schema transformation and integration have to encompass integrity constraints. Global integrity constraints can be used to optimize global queries and to validate global updates. In addition to the transformed local integrity constraints the federated schema contains 'integration constraints' [BC86] stemming from inter-schema correspondence assertions [SPD92]. These integration constraints force global consistency.

The second requirement demands to transform integrity constraints during the federation design process. In our approach, we consider integrity constraints referring to class extensions. A typical schema conflict occurs if extensions of two local classes can overlap. This overlapping itself is an integration constraint. By adding such integration constraints, local integrity constraints can come into conflict. That is, for objects simultaneously stored in both classes different conflicting integrity constraints can be defined. The problem in this case is to find a federated schema fulfilling the requirements mentioned above. Various publications tackle this conflict. [BC86,Con86], for instance, describe the integration of views and name local schemata (views) without conflicting integrity constraints conflictfree schemata. In general, testing for conflictfreeness is undecidable. In those approaches only conflictfree schemata (views) can be integrated correctly. [RPG95,EJ95] show how to integrate schemata into a federated schema considering integrity constraints. Conflicting integrity constraints are not integrated. [VA96] distinguish between subjective and objective constraints. Subjective constraints are only valid in the context of a local database and need not to be dealt with during schema integration. In this way only objective and non-conflicting integrity constraints are integrated.

In [AQFG95] the integration of conflicting integrity constraints bases on a conflict classification. In some cases, the more relaxed constraint is taken over to the integrated class. In other cases, the local constraints are reformulated in an implication format and attached to the integrated class.

In contrast to the mentioned approaches we integrate integrity constraints basing on set operations on class extensions. Extensions of local classes have to be decomposed into disjoint base extensions and later to be composed to the federated schema (cf. [SS96a,SC97,CHS+97]). Local integrity constraints are related to class extensions. This idea is already mentioned shortly in [BCW91]. By decomposing and composing class extensions in a well-defined way, we are able to derive integrity constraints for integrated schemata (as well as for arbitrary views, e.g. external schemata) in a clean and logical sound way. We tackle the problem of conflictfreeness of local schemata in a different way than most other approaches mentioned before. It is possible to discover conflicting integrity constraints during database schema integration. Such conflicts are often a hint that there are other conflicts behind that, e.g. a semantic conflict or a name conflict which has not been discovered up to that moment.

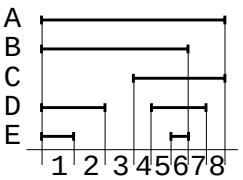
In this paper we focus on the integration of integrity constraints based on the integration of the extensions (classes) they belong to. Thereby, we continue the work presented in [CHS+97] where a basic analysis of the problem of integrating integrity constraints was given. The next section introduces elementary operations for decomposing and composing class extension. For these operations we present rules to transform and integrate integrity constraints. We briefly

discuss the application of these rules to classes of frequently occurring integrity constraints before we give a detailed example.

The rules presented in this paper are not intended to be applied in a fully automatic manner. The designer of the federated database has to control the application of them. In this way he can see during the integration process whether there are conflicting constraints and in which way the integration has to be changed for resolving such conflicts.

2 Elementary Operations

During the schema integration process the main task is to find all relationships (integration constraints) between the class extensions in the base schemata which are to be integrated. Two class extensions can be always equal or always disjoint. In addition to these relationships, two class extensions can overlap semantically. A special case of overlapping class extensions is that one is always a subset of another one (cf. w.r.t. these relationships for instance [SPD92]). As showed in [SS96b], pairwise comparisons of class extensions is not always sufficient because not all pieces of information necessary for an adequate integration can be found in this way. Instead, all class extensions must be considered at the same time. The result of such an analysis can be presented in a graphical manner as depicted as follows:



The horizontal lines represent the class extensions of five classes A, B, C, D, and E, respectively. For instance, the class extensions of B and C are overlapping. Furthermore, E is a subclass of D. Based on this information, a set of *base extensions* can be identified as a canonical representation. In the figure, eight base extensions have been found and numbered by 1 to 8.

Using the standard set operations *intersection* and *difference* we can describe the base extensions in terms of the original class extensions. After such a decomposition into base extensions we can construct each possible class extension which might occur in the integrated schema by applying the standard *union* operation to a collection of base extensions. Based on this simple procedure of decomposing and then composing class extensions, resolving of extensional conflicts can be described in terms of the three standard set operations. In [SS96a] an algorithm is proposed to compose base extensions to a federated schema automatically. Due to the fact that *selection* is a very important operation for defining views or specifying export schemata, we also consider *selection* to be an elementary operation.

The decomposition of class extensions can produce base extensions which belong to local classes from different databases. Objects of such base extensions are managed by more than one database system simultaneously. The values of common attributes are stored redundantly. If the local databases store the current state of real-world objects correctly then the values of the common

attributes would be equal. Otherwise we can state that different values indicate inconsistency.

For two local classes of which the extensions overlap conflicting integrity constraints concerning common attributes can exist. For instance, the attribute salary can be restricted by 'salary $>$ 3000' in one database and by 'salary $>$ 2500' in another database. Such conflicting integrity constraints defined for common attributes of objects stored in both databases are indications of

1. wrong specification of extensional overlappings or
2. wrong specification of attribute-attribute relationships (attribute conflicts) or
3. incomplete or incorrect design of at least one local schema.

A wrong analysis of the schemata to be integrated (first and second item) can be corrected without violation of local autonomy. A wrong design of a local schema is more tricky. If both databases are consistent, i.e. the values of common attributes are equal, then the integrity constraints of both classes hold simultaneously. In this case the local integrity constraints can be corrected without invalidating the existing database. The corrected local integrity constraints would then help to detect wrong local updates.

However, sometimes the local database should continue in possibly inconsistent states due to the demand for design autonomy. A solution then is to regard common attributes as semantically different. In this way, the integrity constraints are not longer in conflict.

The next section presents rules for treating integrity constraints during the decomposition and composition of class extensions.

3 Rules for Dealing with Integrity Constraints

For presenting the rules which can be applied to integrity constraints during schema integration we first need to introduce our notation and to define some important terms.

As already mentioned before, integrity constraints are usually given for a certain set of objects. Therefore, we always consider constraints together with the sets of objects for which they are valid. We denote by $\Theta_\phi(E)$ that the constraint ϕ holds for a set E of objects. Typically, E is a class extension. From a logical point of view, $\Theta_\phi(E)$ says that all free variables occurring in ϕ are bound to the range E. The range E is also called the *validity range* of the constraint ϕ.

For simplifying the presentation we distinguish between two basic kinds of constraints. An *object constraint* is a constraint which can be checked independently for each object in the validity range of that constraint. Integrity checking for object constraints is easy and can be done very efficiently. In case we insert a new object or modify an existing object, we only need to consider that object for checking object constraints.

In contrast, a *class constraint* can be characterized as a constraint which can only be checked by considering at least two objects out of its validity range. Often, all currently existing objects in the validity range are needed for checking a class constraint.[1] Obviously, it is possible to distinguish several different kinds

[1] The distinction between object constraints and class constraints follows the taxonomy of integrity constraints presented in [AQFG95].

of class constraints, but, for the purposes of this paper, we do not need such a distinction.

In the following, we consider the four elementary operations identified in the previous section. For each elementary operation we give a rule for dealing with object constraints. Afterwards, we briefly discuss in which cases these rules can also be applied to class constraints.

Selection. Applying a selection σ_ψ (where ψ is the selection condition) to an extension E results in a subset of E denoted by $\sigma_\psi(E)$. The selection condition ψ can be considered as an object constraint. In consequence, ψ holds for the extension $\sigma_\psi(E)$. Assuming that an object constraint ϕ holds for E, we can conclude that $\phi \wedge \psi$ holds for $\sigma_\psi(E)$:

$$\Theta_\phi(E) \rightarrow \Theta_{\phi \wedge \psi}(\sigma_\psi(E)).$$

Difference. Applying the difference operation results in a subset of the first operand. Therefore, the difference operation and taking a subset can be dealt with in the same way with respect to object constraints. If an object constraint ϕ holds for an extension E, ϕ also holds for each subset of E. This is due to the fact that ϕ is an object constraint which can be checked independently for each object in E and this is true for each subset of E as well. This property leads to the following rule (where $E = E_1 \cup E_2$ and, thus, E_1 is a subset of E):

$$\Theta_\phi(E_1 \cup E_2) \rightarrow \Theta_\phi(E_1).$$

Intersection. The intersection of two (or more) extensions is always a subset of each of the original extensions. Due to the fact that object constraints can be transferred from an extension to any subset of that extension (as explained before), the following rule holds for object constraints. If an object constraint ϕ holds for an extension E_1 and an object constraint ψ holds for an extension E_2, then both ϕ and ψ (and thereby also the logical conjunction $\phi \wedge \psi$) hold for the intersection of E_1 and E_2:

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi \wedge \psi}(E_1 \cap E_2).$$

Union. In case we unite two extensions E_1 and E_2 we can only use the knowledge that each object in $E_1 \cup E_2$ is an object in E_1 or in E_2. In consequence, if ϕ is an object constraint valid for E_1 and ψ an object constraint valid for E_2, then $\phi \vee \psi$ is valid for $E_1 \cup E_2$:

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi \vee \psi}(E_1 \cup E_2).$$

In Figure 1 we have listed all rules introduced so far (rules (1) to (4)). In addition, the rules (5) and (6) express important logical properties.

In case an integrity constraint consists of a conjunction of several conditions, each of these conditions can be considered as a single integrity constraint valid for the same extension (rule (5)). Furthermore, we can weaken each constraint by disjunctively adding an arbitrary further condition (rule (6)). Rule (7) says that two constraints being valid for the same extension may be put together into

Let ϕ and ψ be object constraints, then the following rules hold:

$$\Theta_\phi(E) \rightarrow \Theta_{\phi\wedge\psi}(\sigma_\psi(E)) \quad (1)$$

$$\Theta_\phi(E_1 \cup E_2) \rightarrow \Theta_\phi(E_1) \quad (2)$$

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi\wedge\psi}(E_1 \cap E_2) \quad (3)$$

$$\Theta_\phi(E_1), \Theta_\psi(E_2) \rightarrow \Theta_{\phi\vee\psi}(E_1 \cup E_2) \quad (4)$$

$$\Theta_{\psi\wedge\phi}(E) \rightarrow \Theta_\psi(E) \quad (5)$$

$$\Theta_\psi(E) \rightarrow \Theta_{\psi\vee\phi}(E) \quad (6)$$

$$\Theta_\phi(E), \Theta_\psi(E) \rightarrow \Theta_{\phi\wedge\psi}(E) \quad (7)$$

$$\rightarrow \Theta_{\text{true}}(E) \quad (8)$$

Fig. 1. Overview of rules for *object constraints*

a conjunction which is still valid for that extension. For completeness, rule (8) allows to introduce the constraint true being valid for each extension.

Considering rule (7) we can see that it is only a special case of rule (3) where E_1 equals E_2. Because this is an important case often occurring during schema integration, we give an own rule for that.

In case E_1 equals E_2 we can also apply rule (4). Then, $\phi \vee \psi$ can be derived for E_1 (and E_2, resp.). From a formal point of view this result is correct, although it is too weak. In general, we are interested in obtaining the strongest constraint(s) which can be derived. Nevertheless, the rules allow us to derive weaker constraints.

Up to now, we have only dealt with object constraints. Due to the fact that the rules (5)–(7) describe general logical properties they can be used for arbitrary constraints including class constraints without modification. For the other rules we have now to see in which cases they can be applied to class constraints.

4 Applicability of the Rules to Class Constraints

The rules presented in the previous section hold for object constraints. The questions now is whether these rules are valid for class constraints, too. In this section we investigate certain kinds of class constraints (uniqueness constraints, referential integrity constraints, and constraints based on the aggregate functions max and min) which are relevant in practice.

4.1 Uniqueness Constraints

The rules (1), (2), and (3) also hold for uniqueness constraints (unique). Rule (1) means that all (selected) objects of the resulting set fulfill the uniqueness constraint besides the selection condition. Obviously, uniqueness constraints hold for each subset of a class extension for which they are originally defined. This follows immediately from the fact that an object which can be unambiguously identified in a class extension can also be unambiguously identified in each subset of this class extension. The rules (2) and (3) also hold for the same reason.

However, the following aspect has to be considered for all three rules: A uniqueness constraint which is originally defined on a superset is generally weakened by using the rules (1), (2), and (3). We will illustrate this aspect by using an example: Suppose, there is a class extension employees for which a uniqueness

constraint is defined. Further assume that this class extension can be decomposed into two base extensions project group 1 and project group 2. In this case, we can state that the uniqueness constraint holds for both base extensions. Nevertheless, the "decomposition" of the uniqueness constraint leads to a weakening because the relationship between the base extensions is not considered. Now, it is possible that in both base extensions an object can be inserted with the same key attribute values without violating one of the two derived uniqueness constraints. However, from the point of view of the original class extension the originally defined uniqueness constraint is violated.

Thus, we require a further rule which maintain the original integrity constraints. Consequently, we have to introduce the following *equivalence* rule

$$\Theta_\phi(E) \leftrightarrow \Theta_\phi(E_1 \cup E_2) \tag{9}$$

which says that a uniqueness constraint ϕ which holds for an extension E ($E \equiv E_1 \cup E_2$) also holds for the union of the both decomposed extensions E_1 and E_2 (and vice versa). The equivalence rule is not restricted to uniqueness constraints only; obviously, it holds for all kinds of constraints.

The rule (4) does not hold for uniqueness constraints on class extensions because this kind of constraints can be violated by the objects of the second class extension to be united. Thus, we cannot conclude that a uniqueness constraint holds for the united extension. This is also the case when in both class extension the same uniqueness constraint is defined (i.e. unique defined on the same attribute or attribute combination).

Let us consider the following example to illustrate why the rule (4) cannot be applied to uniqueness constraints: Suppose, there are two extensions project group 1 and project group 2. On each of these extensions there is a uniqueness constraint defined. The employees which are involved in the project group 1 are unambiguously identified by their social security number whereas the employees of project group 2 are distinguished using their names. How can these original integrity constraints be handled after unifying the corresponding class extensions? Obviously, we cannot assume that one of the original constraints holds for the whole united extension. On the other hand, these constraints cannot be skipped because of the requirement that the federated schema has to be complete in order to maintain the semantics of the original schemata to be integrated.

This problem can be solved by using *discriminants* [GCS95,AQFG95]. Discriminants can be used to maintain the relationship of the objects of the united extension with the original class extensions they stem from. For the example described above we can define a discriminant attribute group which can be used to specify integrity constraints on the united extension. For instance, the following integrity constraints can be derived in our example: All employee objects whose discriminant has the value 'project group 1' must have different social security numbers. Analogously, all employee objects whose discriminant has the value 'project group 2' must have different names.

4.2 Referential Integrity Constraints

Here, we restrict the discussion of the applicability of the rules for referential integrity constraints to constraints which refer to object references within a single

extension. In our example, there may be a referential integrity constraint on the class extension employees which defines that the attribute supervisor must refer to an existing employee object.

We state that the rule (1) does not hold for such kinds of constraints. This is due to the fact that the result of a selection operation is a subset of the original extension, i.e., some of the objects of the original extension (which do not fulfill the selection condition) are not in the resulting extension. Thus, originally valid references can now be undefined. In other words, we cannot assume that a referential integrity constraint holds for the resulting extension if potentially referenced objects are not in the resulting extension. The rules (2) and (3) do not hold for the same reason.

In contrast, the union rule (4) holds for referential integrity constraints. The reason is that all references of the original extension stay valid because all objects of this extension are in the united extension. Suppose, we unite the extensions project group 1 and project group 2 to the extension employee. Here, we assume that all references are valid because all referenced employees are in the united extension. However, we have to notice that the logical disjunction of the two original integrity constraints leads to a weakening of the constraints. As discussed before, a discriminant approach can be used to solve this problem.

4.3 Aggregate Integrity Constraints

Finally, we discuss the applicability of the rules to aggregate constraints defined by using the aggregate functions max and min. Here, we consider constraints of the form '$\mathsf{max}(\mathsf{x})\ \theta\ \mathsf{z}$' and '$\mathsf{min}(\mathsf{x})\ \theta\ \mathsf{z}$', respectively, whereby x is an attribute, z a constant value and θ is a comparison operator.

In general, the applicability of the rules to such kinds of constraints[2] depends on the comparison operator θ used for the formulation of such kinds of constraints.

For max-constraints it can be easily shown that the rules (1), (2) and (3) can only be applied if the following holds for the comparison operator: $\theta \in \{\leq, <\}$. If the maximum of an attribute x of an extension E is less (or less equal) than a certain value z, then the maximum of the attribute x of each subset of the extension E is also less (or less equal) than the value z. However, in case a max-constraint is defined with one of the following comparison operators $\{\geq, >, =, \neq\}$, then we cannot guarantee that the maximum of the resulting extension fulfills this constraint. Suppose, there is an extension employees on which the max-constraint '$\mathsf{max}(\mathsf{salary}) > 4000$' is defined. Here, it can happen that the object which make the condition valid in the original extension is not available in the resulting extension. Obviously, this is the case when, for instance, the selection condition is '$\mathsf{max}(\mathsf{salary}) \leq 4000$'.

Analogously, the rules (1), (2) and (3) can only be applied to min-constraints if the comparison operator θ is in $\{\geq, >\}$.

The rule (4) is obviously applicable to max-constraints if the comparison operator θ is in $\{\geq, >\}$ and for min-constraints if the comparison operator θ is in $\{\leq, <\}$.

[2] In the following, aggregate constraints based on the function max are called max-constraints. Analogously, aggregate constraints based on the function min are called min-constraints.

Assume, there is a constraint '$\mathsf{max(salary)} \geq 4000$' defined on a class extension. Then, we can assume that this constraint also holds for union with another extension because this constraint is already fulfilled by the original extension. In this case, union means that additional objects (of a second extension) are put in this extension. Therefore, the maximum salary of the united extension can only be greater or equal than the maximum salary of the original extension.

Analogously, assume that there is a constraint '$\mathsf{min(salary)} \leq 2000$' defined on a class extension. This constraint also holds for union because the minimum salary of the united extension can only be lower or equal than the minimum salary of the original extension.

However, there are two other cases where rule (4) is applicable, too. Firstly, it holds for max-constraints if the comparison operator θ is in $\{\leq, <\}$. Secondly, this rule can be applied to min-constraints if the comparison operator θ is in $\{\geq, >\}$. The reason for the applicability in both cases is that these kinds of constraints can be transformed into object constraints. The class constraint '$\mathsf{max(x)}\ \theta_1\ \mathsf{z}$' with $\theta_1 \in \{\leq, <\}$ is equivalent to the object constraint '$\mathsf{x}\ \theta_1\ \mathsf{z}$' and the class constraint '$\mathsf{min(x)}\ \theta_2\ \mathsf{z}$' with $\theta_2 \in \{\geq, >\}$ corresponds to the object constraint '$\mathsf{x}\ \theta_2\ \mathsf{z}$'. Due to the fact that the rule (4) is applicable to any object constraint we can conclude that this rule can also be applied to these special kinds of max- and min-constraints.

As discussed before, a discriminant is required in order to avoid the weakening of the constraints by applying rule (4). However, the results of the applicability of the rules to class constraints are summarized in the following table:

Rule	**unique**	**ref**	$\mathbf{max}_\theta$	$\mathbf{min}_\theta$
(1) *selection*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(2) *difference*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(3) *intersection*	$\surd$	—	$\theta \in \{\leq, <\}$	$\theta \in \{\geq, >\}$
(4) *union*	—	$\surd$	$\theta \in \{\geq, >, <, \leq\}$	$\theta \in \{\leq, <, \geq, >\}$

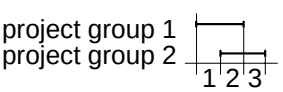
5 Example for Rule Application

In this section we demonstrate and discuss the application of our proposed rules by means of a small example. For the presentation, we distinguish two phases:

1. decomposing the original extensions into base extensions, and
2. composing these base extensions by uniting them to derived extensions.

5.1 Decomposing Extensions

The decomposition of the original class extensions is based on an extensional analysis. Such an analysis specifies in which way we have to construct the base extensions from the original extensions by applying set operations. In our example, we start with two class extensions E_1 and E_2 which contain employees working in two different projects groups. We assume that the result of the extensional analysis is as it is depicted in the following figure:



There may be employees working only for one of the two projects, and there are employees working for both projects at the same time. In consequence, the class extensions overlap.
Now, we decompose the original class extensions into three base extensions. The decomposition can be described using the set operations intersection and difference between E_1 and E_2:

$$E_1' = E_1 - E_2, \quad E_2' = E_1 \cap E_2, \quad E_3' = E_2 - E_1$$

In Figure 2 the resulting three base extensions are placed on the second level above the two original class extensions.

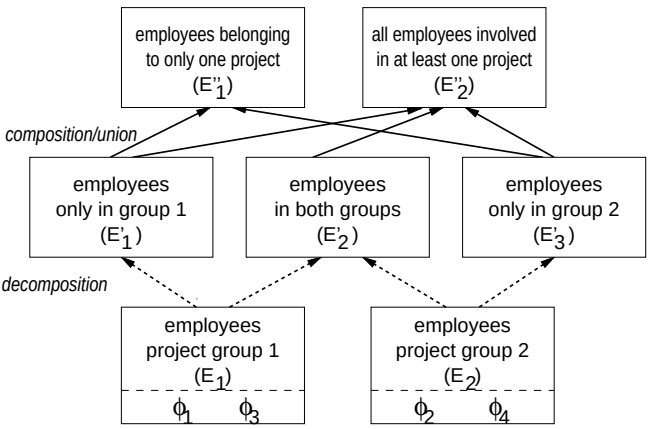


Fig. 2. Decomposing and uniting extensions of the example

Now, let us assume that we have the following integrity constraints imposed on the two original class extensions:

$$\Theta_{\phi_1}(E_1) \; with \; \phi_1 \equiv \forall e : e.\text{qualification} > 4$$
$$\Theta_{\phi_3}(E_1) \; with \; \phi_3 \equiv \forall e_1, e_2 : e_1.\text{pid} = e_2.\text{pid} \Longrightarrow e_1 = e_2$$
$$\Theta_{\phi_2}(E_2) \; with \; \phi_2 \equiv \forall e : e.\text{qualification} > 3$$
$$\Theta_{\phi_4}(E_2) \; with \; \phi_4 \equiv \exists e_1 : \forall e_2 : e_1.\text{salary\_level} \geq e_2.\text{salary\_level} \wedge e_1.\text{salary\_level} > 3$$

ϕ_1 and ϕ_2 are object constraints whereas ϕ_3 and ϕ_4 are class constraints. ϕ_3 requires the uniqueness of values for the attribute pid (i.e., pid can be used as a key). ϕ_4 is an aggregate constraint concerning the maximum value for the attribute salary_level (we may use the short term max(salary_level)>3). In other words there must be at least one employee of extension E_2 having a higher salary level than 3.

Using the relationships between base extensions and original class extensions we can derive integrity constraints for the base extensions. For that we have to

consider each constraint and to apply our rules. The object constraints ϕ_1 and ϕ_2 do not cause any problem. By applying rule (2) and (9) we derive:

$$\Theta_{\phi_1}(E_1) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_1}(E_1' \cup E_2') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_1}(E_1')$$
$$\Theta_{\phi_1}(E_1) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_1}(E_2' \cup E_1') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_1}(E_2')$$
$$\Theta_{\phi_2}(E_2) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_2}(E_2' \cup E_3') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_2}(E_2')$$
$$\Theta_{\phi_2}(E_2) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_2}(E_3' \cup E_2') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_2}(E_3')$$

Transferring the class constraints from the original extensions to the base extensions is more complicated. For instance, the application of rule (2) to ϕ_3 leads to a weakening of the constraint. As long as we do not forget the original constraint, this is not a real problem:

$$\Theta_{\phi_3}(E_1) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_3}(E_1' \cup E_2') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_3}(E_1')$$
$$\Theta_{\phi_3}(E_1) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_3}(E_2' \cup E_1') \overset{\text{rule } 2}{\longrightarrow} \Theta_{\phi_3}(E_2')$$
$$\Theta_{\phi_3}(E_1) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_3}(E_1' \cup E_2')$$

In contrast to the uniqueness constraint ϕ_3, the max-constraint ϕ_4 cannot be transferred to any base extension. Hence, only rule (9) can be applied:

$$\Theta_{\phi_4}(E_2) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_4}(E_2' \cup E_3')$$

5.2 Uniting Extensions

For the decomposition of extensions the respective rules are applied to each *single* constraint. However, for the composition the application of rule (4) to each simple constraint is not useful. In order to avoid a constraint weakening, rule (4) has to be applied to *all* possible combinations of constraints and afterwards the results have to be conjunctively combined by rule (7). A simpler way to deal with constraints is to conjunctively combine the constraints of each base extension before applying other rules.

In our integration example the federated schema consists of the class extensions E_1'' and E_2''. Extension E_1'' results from uniting base extensions E_1' and E_3' and contains all employees working in only one project group whereas extension E_2'' contains all employees. The class E_1'' is a subclass of class E_2''. This federated schema is only one possible federated schema. However, it demonstrates the application of our proposed rules.

The constraints of base extensions E_1' and E_2' have to be combined by applying rule (7). On base extension E_3' only one constraint holds. Therefore, the constraints of the base extensions E_1' and E_2' only have to be combined:

$$\Theta_{\phi_1}(E_1'), \Theta_{\phi_3}(E_1') \overset{\text{rule } 7}{\longrightarrow} \Theta_{\phi_1 \wedge \phi_3}(E_1')$$
$$\Theta_{\phi_1}(E_2'), \Theta_{\phi_2}(E_2'), \Theta_{\phi_3}(E_2') \overset{\text{rule } 7}{\longrightarrow} \Theta_{\phi_1 \wedge \phi_2 \wedge \phi_3}(E_2') \leftrightarrow \Theta_{\phi_1 \wedge \phi_3}(E_2')$$

Here, constraint ϕ_2 can be omitted since it is weaker than constraint ϕ_1.

$\mathbf{E_1''}$: The rule (4) disjunctively brings together the combined constraints of the base extensions E_1' and E_3':

$$\Theta_{\phi_1 \wedge \phi_3}(E_1'), \Theta_{\phi_2}(E_3') \xrightarrow{\text{rule } 4} \Theta_{(\phi_1 \wedge \phi_3) \vee \phi_2}(E_1' \cup E_3') \leftrightarrow \Theta_{\phi_2}(E_1' \cup E_3') \xrightarrow{\text{rule } 9} \Theta_{\phi_2}(E_1'')$$

Due to the fact that $\phi_1 \Rightarrow \phi_2$ holds, the formula $(\phi_1 \wedge \phi_3) \vee \phi_2$ can be simplified to ϕ_2. Neither constraint ϕ_3 nor ϕ_4 holds on E_1''. Objects can be inserted into extension E_1'' which cannot be propagated to E_1 nor E_3 because both class constraints are violated. It is impossible to test these class constraints for extension E_1'' due to the missing base extension E_2' on the composition level. Errors from violating update operations cannot be interpreted on this level. This problem is similar to the well-known *view-update*-problem of selective views.

$\mathbf{E_2''}$: The extension E_2'' can be formed by:

$$E_2'' = E_1' \cup E_2' \cup E_3' = (E_1' \cup E_2') \cup (E_2' \cup E_3')$$

The integrity constraints of class extension $E_1' \cup E_2'$ cannot be derived directly by applying rule (4) due to the class constraint ϕ_3. This constraint has to be weakened before:

$$\Theta_{\phi_1 \wedge \phi_3}(E_1') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_1') \qquad \Theta_{\phi_1 \wedge \phi_3}(E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_2')$$

The application of rule (4) produces:

$$\Theta_{\phi_1}(E_1'), \Theta_{\phi_1}(E_2') \xrightarrow{\text{rule } 4} \Theta_{\phi_1 \vee \phi_1}(E_1' \cup E_2') \leftrightarrow \Theta_{\phi_1}(E_1' \cup E_2')$$

By applying rule (7) we obtain:

$$\Theta_{\phi_1}(E_1' \cup E_2'), \Theta_{\phi_3}(E_1' \cup E_2') \xrightarrow{\text{rule } 7} \Theta_{\phi_1 \wedge \phi_3}(E_1' \cup E_2')$$

Analogously to $E_1' \cup E_2'$, we have to weaken the constraint before applying rule (4):

$$\Theta_{\phi_1 \wedge \phi_3}(E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_2')$$
$$\Theta_{\phi_1}(E_2'), \Theta_{\phi_2}(E_3') \xrightarrow{\text{rule } 4} \Theta_{\phi_1 \vee \phi_2}(E_2' \cup E_3') \leftrightarrow \Theta_{\phi_2}(E_2' \cup E_3')$$

Due to the fact that $\phi_1 \Rightarrow \phi_2$ holds, the formula $\phi_1 \vee \phi_2$ can be simplified to ϕ_2. By applying rule (7) we obtain:

$$\Theta_{\phi_2}(E_2' \cup E_3'), \Theta_{\phi_4}(E_2' \cup E_3') \xrightarrow{\text{rule } 7} \Theta_{\phi_2 \wedge \phi_4}(E_2' \cup E_3')$$

Before the extension $E_1' \cup E_2'$ and $E_2' \cup E_3'$ can be united by applying rule (4) the constraint ϕ_3 has to be removed:

$$\Theta_{\phi_1 \wedge \phi_3}(E_1' \cup E_2') \xrightarrow{\text{rule } 5} \Theta_{\phi_1}(E_1' \cup E_2')$$

By applying rule (4) we obtain:

$$\Theta_{\phi_1}(E_1' \cup E_2'), \Theta_{\phi_2 \wedge \phi_4}(E_2' \cup E_3') \xrightarrow{\text{rule } 7} \Theta_{\phi_1 \vee (\phi_2 \wedge \phi_4)}(E_1' \cup E_2') \xrightarrow{\text{rule } 9} \Theta_{\phi_1 \vee (\phi_2 \wedge \phi_4)}(E_2'')$$

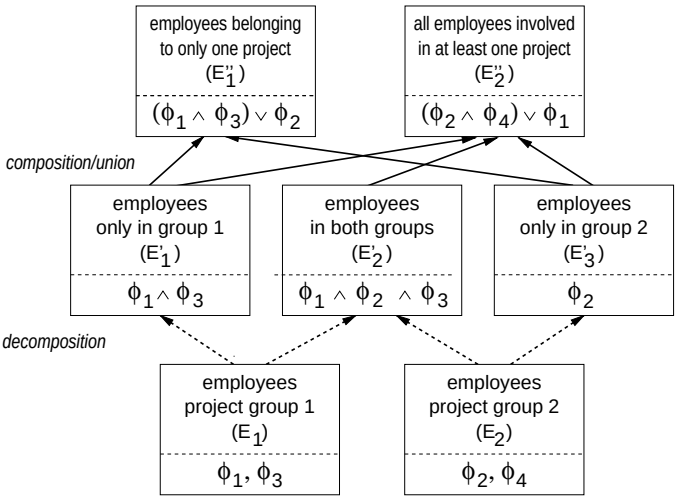


Fig. 3. Extensions with derived integrity constraints

Figure 3 shows the extensions with their integrity constraints in the unreduced form.

An interesting point is the fact that the uniqueness constraint ϕ_3 disappeared from extension E_1'' due to the disjunctive combination expressed in rule (4). The uniqueness constraint does not hold for the extension E_2'', too. The introduction of an artificial discriminant attribute avoids such an disappearance. The discriminant attribute (e.g. group) in our example expresses the information whether an employee works for project group 1 (value 1) or project group 2 (value 2) or for both of them (value 3). This information can be used within the constraints in form of an implication. The left hand side of the implication tests the discriminant attribute on a specific value and the right hand side contains the original constraint. We transform the constraints ϕ_1, ϕ_2, ϕ_3, and ϕ_4 to $\phi_1', \phi_2', \phi_3'$, and ϕ_4', respectively, and assign them to the extensions E_1', E_2', and E_3'.

$$\phi_1' \equiv \forall e : (e.\text{group} = 1 \vee e.\text{group} = 3) \Longrightarrow e.\text{qualification} > 4$$
$$\phi_2' \equiv \forall e : (e.\text{group} = 2 \vee e.\text{group} = 3) \Longrightarrow e.\text{qualification} > 3$$
$$\phi_3' \equiv \forall e_1, e_2 : ((e_1.\text{group} = 1 \vee e_1.\text{group} = 3) \wedge (e_2.\text{group} = 1 \vee e_2.\text{group} = 3)) \Longrightarrow (e_1.\text{pid} = e_2.\text{pid} \Longrightarrow e_1 = e_2)$$
$$\phi_4' \equiv \exists e_1 : \forall e_2 : ((e_1.\text{group} = 2 \vee e_1.\text{group} = 3) \wedge (e_2.\text{group} = 2 \vee e_2.\text{group} = 3)) \Longrightarrow e_1.\text{salary\_level} \geq e_2.\text{salary\_level} \wedge e_1.\text{salary\_level} > 3$$

$$\Theta_{\phi_1' \wedge \phi_3'}(E_1'), \qquad \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_2'), \qquad \Theta_{\phi_2'}(E_3')$$

These transformed constraints can now be conjunctively combined. The conjunctive combination stands in contrast to rule (4).

$$\Theta_{\phi_1' \wedge \phi_3'}(E_1'), \Theta_{\phi_2'}(E_3') \longrightarrow \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_1' \cup E_3') \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi_1' \wedge \phi_2' \wedge \phi_3'}(E_1'')$$

$$\Theta_{\phi'_1 \wedge \phi'_3}(E'_1), \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3}(E'_2), \Theta_{\phi'_2}(E'_3), \Theta_{\phi'_4}(E'_2 \cup E'_3)$$
$$\longrightarrow \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4}(E'_1 \cup E'_2 \cup E'_3) \overset{\text{rule } 9}{\longrightarrow} \Theta_{\phi'_1 \wedge \phi'_2 \wedge \phi'_3 \wedge \phi'_4}(E''_2)$$

Introducing discriminant attributes and respective transformation of constraints reduce weakening of constraints during schema integration (or more general during schema restructuring).

6 Conclusions

The complete treatment of integrity constraints during the schema integration is a fundamental requirement, e.g., for global integrity enforcement in federated database systems. Moreover, the extensive consideration of integrity constraints plays an important role in application scenarios such as view materialization [SJ96] or mobile databases [WSD+95].

In this paper, we presented an approach to handle integrity constraints during the schema integration. We found a small set of general rules which specify how to deal with integrity constraints during the integration process. We showed that these rules can be used for object constraints without any restrictions. For several classes of often occurring class constraints we presented the applicability of the rules. Of course, there are some logical restrictions for which specialized solutions are needed. Beside the results presented here, we have already investigated further classes of constraints, e.g. constraints with other aggregate functions.

Furthermore, we presented a systematic application of the rules. By means of a running example we sketched some critical problems such the weakening of constraints and demonstrated how to handle these kinds of problems (e.g. by using discriminants).

Our approach is not limited to schema integration purposes only, it can also be used for view derivation and arbitrary schema modification operations such as schema transformation or schema restructuring. We are currently working on the completion of the classification of integrity constraints and the applicability of our rules to further classes of constraints. Using a classification which is based on the logical structure of constraints (assuming an adequate normal form for the logical representation of constraints) seems to be a very promising approach for completing the formal part of our work.

Finally, we can state that beside our schema integration strategies presented in [SS96a,SS96b,SC97], the consideration of integrity constraints is another step towards a unified integration methodology.

References

AQFG95. R. M. Alzahrani, M. A. Qutaishat, N. J. Fiddian, and W. A. Gray. Integrity Merging in an Object-Oriented Federated Database Environment. In C. Goble and J. Keane, eds., *Advances in Databases, Proc. BNCOD-13*, LNCS 940, pages 226–248. Springer, 1995. 120, 122, 125

BC86. J. Biskup and B. Convent. A Formal View Integration Method. In C. Zaniolo, ed., *Proc. 1986 ACM SIGMOD Int. Conf. on Management of Data*, ACM SIGMOD Record, 15(2):398–407, 1986. 120, 120

BCW91. M. Bouzeghoub and I. Comyn-Wattiau. View Integration by Semantic Unification and Transformation of Data Structures. In H. Kangassalo, ed., *Entity-Relationship Approach: The Core of Conceptual Modelling, Proc. ER'90*, pages 381–398. North-Holland, 1991. 120

CHS+97. S. Conrad, M. Höding, G. Saake, I. Schmitt, and C. Türker. Schema Integration with Integrity Constraints. In C. Small, P. Douglas, R. Johnson, P. King, and N. Martin, eds., *Advances in Databases, Proc. BNCOD-15*, LNCS 1271, pages 200–214. Springer, 1997. 120, 120

Con86. B. Convent. Unsolvable Problems Related to the View Integration Approach. In G. Ausiello and P. Atzeni, eds., *Proc. 1st Int. Conf. Database Theory (ICDT'86)*, LNCS 243, pages 141–156. Springer, 1986. 120

EJ95. L. Ekenberg and P. Johannesson. Conflictfreeness as a Basis for Schema Integration. In S. Bhalla, ed., *Information Systems and Data Management, Proc. CISMOD'95*, LNCS 1006, pages 1–13. Springer, 1995. 119, 120

GCS95. M. Garcia-Solaco, M. Castellanos, and F. Saltor. A Semantic-Discriminated Approach to Integration in Federated Databases. In S. Laufmann, S. Spaccapietra, and T. Yokoi, eds., *Proc. 3rd Int. Conf. on Cooperative Information Systems (CoopIS'95)*, pages 19–31, 1995. 125

LNE89. J. A. Larson, S. B. Navathe, and R. Elmasri. A Theory of Attribute Equivalence in Databases with Application to Schema Integration. *IEEE Transactions on Software Engineering*, 15(4):449–463, 1989. 119

RPG95. M. P. Reddy, B. E. Prasad, and A. Gupta. Formulating Global Integrity Constraints during Derivation of Global Schema. *Data & Knowledge Engineering*, 16(3):241–268, 1995. 119, 120

SC97. I. Schmitt and S. Conrad. Restructuring Class Hierarchies for Schema Integration. In R. Topor and K. Tanaka, eds., *Database Systems for Advanced Applications '97, Proc. DASFAA'97*, pages 411–420, World Scientific, 1997. 120, 132

SJ96. M. Staudt and M. Jarke. Incremental Maintenance of Externally Materialized Views. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 75–86. Morgan Kaufmann, 1996. 132

SL90. A. P. Sheth and J. A. Larson. Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases. *ACM Computing Surveys*, 22(3):183–236, 1990. 119

SPD92. S. Spaccapietra, C. Parent, and Y. Dupont. Model Independent Assertions for Integration of Heterogeneous Schemas. *The VLDB Journal*, 1(1):81–126, 1992. 119, 120, 121

SS96a. I. Schmitt and G. Saake. Integration of Inheritance Trees as Part of View Generation for Database Federations. In B. Thalheim, ed., *Conceptual Modelling — ER'96*, LNCS 1157, pages 195–210. Springer, 1996. 119, 120, 121, 132

SS96b. I. Schmitt and G. Saake. Schema Integration and View Generation by Resolving Intensional and Extensional Overlappings. In K. Yetongnon and S. Hariri, eds., *Proc. 9th ISCA Int. Conf. on Parallel and Distributed Computing Systems (PDCS'96)*, pages 751–758. International Society for Computers and Their Application, Releigh, NC, 1996. 121, 132

VA96. M. W. W. Vermeer and P. M. G. Apers. The Role of Integrity Constraints in Database Interoperation. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., *Proc. 22nd Int. Conf. on Very Large Data Bases (VLDB'96)*, pages 425–435. Morgan Kaufmann, 1996. 120

WSD+95. O. Wolfson, P. Sistla, S. Dao, K. Narayanan, and R. Raj. View Maintenance in Mobile Computing. *ACM SIGMOD Record*, 24(4):22–27, 1995. 132

Scoped Referential Transparency in a Functional Database Language with Updates

P.F.Meredith and P.J.H King

Dept. of Computer Science, Birkbeck College, Malet St, London WC1E 7HX
{P.F.Meredith@btinternet.com, pjhk@dcs.bbk.ac.uk}

Abstract. We describe in brief a lazy functional database language *Relief*, which supports an entity-function model and provides for update and the input of data by means of functions with side-effects. An *eager let* construct is used within the lazy graph reduction mechanism to sequence the effects. To redress the loss of referential transparency we have implemented an effects checker which can identify referentially transparent regions or scopes within *Relief*.

1 Introduction

An inherent conflict arises with functional languages in the database context between change of state on update and the concept of referential transparency. Passing the database itself as a parameter resolves the issue conceptually but is not a practical approach. Update by side-effect is efficient but necessarily implies loss of referential transparency. Additionally side-effects are often incompatible with lazy evaluation since the occurrence and sequence of updates can become non-intuitive. Our approach is to accept that a change of database state will result in loss of global referential transparency but that referential transparency can be scoped so that areas of a program can be identified which retain the property. Equational reasoning and consequent optimisation can then be applied within these scopes.

We describe a variant of the functional database language FDL [Pou88, PK90], called *Relief*, which is lazy and which has been extended with functions to facilitate updates and reading of files. These functions work by means of side-effects which are explicitly sequenced with an eager let expression and a derivative sequencing construct. There is no change in the overall laziness of the graph reduction which we illustrate with an example update program which employs the lazy stream model. To redress the loss of global referential transparency an effects checker identifies referentially transparent regions or scopes.

2 The Entity-Function Model

The development of *Relief* is a contribution to the work of the Tristarp Group, initiated in 1984 to investigate the feasibility of a database management system which

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 134-148, 1998.

supports the binary relational model both at the storage and conceptual levels. A summary of the early work is given by King et al. [KDPS90]. More recently research has focused on supporting the entity-function model, a hybrid of the binary relational and functional data models [Shi81, AK94]. The relationships between entities are binary and are represented by (possibly) multivalued single argument functions. Entity types are classified into lexical and non-lexical as with the analysis methodology NIAM [VV82]. Members of lexical types have direct representation (for example the types string and integer) whereas those of non-lexical types are real or abstract objects without direct representation eg. persons, invoices, courses etc.

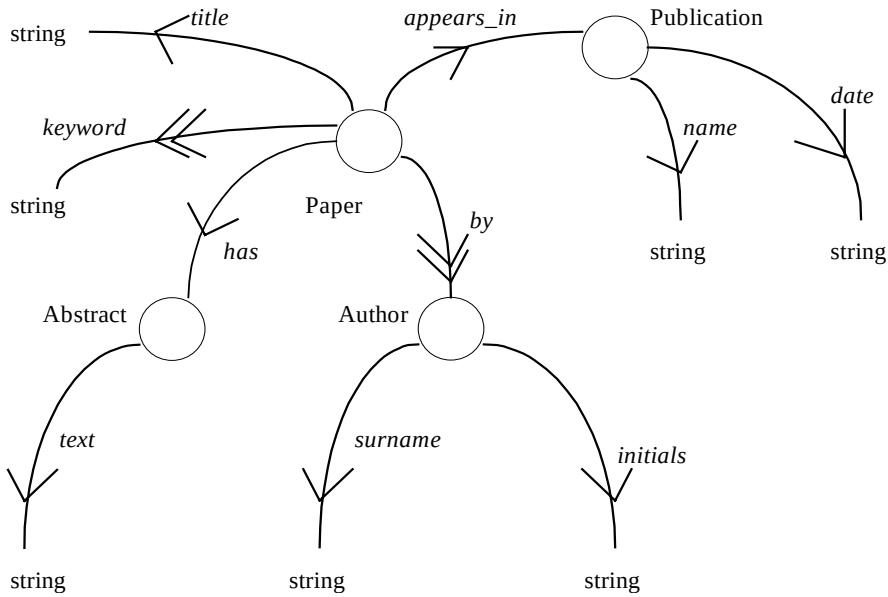


Fig. 1. The Computer Literature Database (CLDB)

Figure 1 is a subschema of the Computer Literature Database (CLDB), a body of data used experimentally by the Tristarp Group, and illustrates the entity-function model. Non-lexical types are shown as circles with functions represented by arcs. A single-headed arrow represents a single-valued function and a double-headed one denotes a multivalued function. Thus a Paper has a single Abstract but is *by* (possibly) several Authors. In this example, the lexical type “string” is used extensively in the ranges of the functions.

3 Defining a Database in *Relief*

Relief is an experimental variant of FDL [Pou88, PK90], the first language to integrate the entity-function model with the functional computation model over a

binary relational storage structure. FDL has the usual features of a modern functional programming language such as polymorphic types, lazy evaluation and pattern-matching and we refer the reader to Field and Harrison [FH88] for an explanation of these. We describe here only the features of *Relief* which illustrate the scoping of referential transparency. A full syntax and description of the language and its implementation is given by Meredith [Mer98]. The persistent functions and entities which constitute a database in *Relief* are held in a triple store, a development of the grid file [NHS84, Der89] which has recently been adapted to hold long strings and to support interval queries.

CLDB contains the non-lexical types *paper* and *author* amongst others. This part of the schema is implemented in *Relief* with the following type declarations:

```
paper :: nonlex;
author :: nonlex;
by : paper -> [author];
title : paper -> string;
surname : author -> string:
initials : author -> string;
```

In order to facilitate easy navigation of the schema *Relief* automatically maintains a converse function for all functions whose domain is a non-lexical type, referred to by the prefix *rev*. For example the converses of the functions *surname* and *by* are:

```
rev_surname : string -> [author];
rev_by : author -> [ paper ];
```

As an example of their use consider the following list comprehension query which retrieves the papers by authors with the surname “Meredith”.

```
[ title p | for a in rev_surname "Meredith";
  for p in rev_by a] ;
```

It is interesting to note here the syntactic similarity between *Relief* and the “FOR EACH” statement in DAPLEX [Shi81].

4 The Four Update Functions of *Relief*

The extent of a database is defined and can be redefined by means of the functions *create*, *destroy*, *define* and *delete*. The function *create* takes a non-lexical type as an argument and returns a new entity of the specified type. For instance:

```
create author;
```

returns a new member of the type *author*. An attempt to print a non-lexical entity yields its type in angled brackets, thus: <*author*>. The extent of a non-lexical type can be queried with the *All* function which returns the members of the specified type. For example, `All author ;`

The function *destroy* destroys an entity of the specified non-lexical type. Thus to destroy the first element in the list of all authors:

```
destroy author (head (All author));
```

In the functional model it is necessary for every expression to have a value and thus destroy returns the value *Done* which is the only value in the type *update*. Functions are incrementally defined using the function *def* which adds an equation to the database and returns the value *Done*. To illustrate this, we give an expression which creates an author entity, binds it to a scoped static variable "a" and then defines an equation for their initials. The binding is done with an eager form of the let expression which is further explained in the next section. Thus:

```
e_let a = create author in (def initials a => "PF");
```

The left hand side of a function equation consists of the name of the function followed by one or more arguments, and the right hand side is an expression which is stored unevaluated. The arguments at definition time can either be patterns or a bound variable such as "a", in which case the value of the variable is used in the definition of the equation. The value of a function's application defaults to the constant "@" (denoting "undefined") if there is no equation which matches the function's arguments. The pattern-matching in *Relief* uses the left-to-right best-fit algorithm developed by Poulovasillis [Pou92].

The delete function *del* deletes an equation from the database and returns the value *Done*. The arguments of *del* identify the left hand side of the function equation for deletion, for example:

```
e_let a = head(rev_surname "King") in (del initials a);
```

deletes the equation defining the initials of an author with the surname "King".

All four update functions are statically type checked, see Meredith [Mer98] for details, and work at the object level. Thus their invocation needs to be sequenced and we discuss this next.

5 Sequencing the Updates

A new construct, the *eager let* (e_let), is introduced to sequence updates by employing call-by-value parameter substitution but within a general scheme of lazy evaluation. We illustrate the eager let expression with the following example which evaluates an expression before using it as the value of the right hand side in an equation definition. The example assumes that people have a unique name and the types of the functions *name* and *salary* are:

```
name : person -> string;
salary : person -> integer;
```

The expression below defines the salary of Bill to be 5000 more than Fred's current salary.

```
e_let bill = head(rev_name "Bill") in
    e_let fred = head(rev_name "Fred") in
          e_let bills_salary = salary fred + 5000 in
                (def salary bill => bills_salary );
```

Thus the value of Bill's salary is stored as an integer and not a formula and so a subsequent change to Fred's salary will not result in a change to Bill's salary. The next example produces a sequence of updates to create an author entity and define *surname* and *initials* before returning the new entity. Two dummy variables, *dum1* and *dum2*, are used to bind with the values returned by calls to the define function.

```
e_let a = create author in
    e_let dum1 = (def surname a => "Meredith") in
          e_let dum2 = (def initials a => "PF")in a;
```

The overall evaluation scheme is still normal order reduction, the outermost let being reduced before the innermost let and the body of each eager let being evaluated lazily. The syntax of the eager let is not ideal because of the need to specify binding patterns which are of no further interest. Hence we have provided an explicit sequencing construct in *Relief* which, although formally only syntactic sugar for the eager let, is much more succinct and convenient for programming. This construct is written as a list of expressions separated by semicolons, enclosed in curly brackets, and is optionally terminated by the keyword *puis* and a following expression which gives a functional value for the entire sequence. The expressions between the brackets are evaluated in their order, from left to right. For example, the expression above can be written as the sequence:

```
  { a = create author; (def surname a=>"Meredith");
(def initials a => "PF") } puis a;
```

We refer to the keyword puis and its following expression as the *continuation*. If it is omitted then a default value of *Done* is inferred for the sequence.

6 An Example - Updating the Computer Literature Database

We describe here a *Relief* program which processes a list of data constructors representing the records in a file "Tristarp_papers" and updates that part of the CLDB concerning authors and papers. *Relief* contains a *read_file* function which can lazily convert a file of records into such a list but space precludes a full description here. The elements of the list are members of the *paper_record* sum type which is defined by the two constructors TITLE_REC and AUTHOR_REC. TITLE_REC has a single string component, whereas AUTHOR_REC has two components: one for the surname of the author and one for the initials:

```
paper_record :: sum;
TITLE_REC : string -> paper_record;
AUTHOR_REC : string string -> paper_record;
```

Our example will assume the following list of records as input:
[(TITLE_REC "Scoped Referential Transparency in a Functional Database Language with Updates"), (AUTHOR_REC "Meredith" "PF"), (AUTHOR_REC "King" "PJH"), (TITLE_REC "TriStarp - An Investigation into the Implementation and Exploitation of Binary Relational Storage Structures"), (AUTHOR_REC "King" "PJH"), (AUTHOR_REC "Derakhshan" "M"), (AUTHOR_REC "Poulovassilis" "A"), (AUTHOR_REC "Small" "C")]

The program to process the list consists of two functions: *cldbup* which creates the paper entities, and *add_authors*, which creates the author entities and links them to the papers. Both functions operate recursively and process records at the head of an input list before returning the remainder of the list. For a list with a title record at its head, cldbup creates a paper, defines its title and calls add_authors as specified by the first equation below. If the head of the list is not a title record then the list is returned unaltered as given by the second equation. This is an appropriate action when an unexpected or unrecognised record is encountered. The third equation specifies that cldbup returns the empty list when given the empty list. Thus cldbup will return the empty list if all of the records in the file can be processed, otherwise it returns some remainder of the input list.

```
cldbup : [paper_record] -> [paper_record];
def cldbup ((TITLE_REC t) : xs) => {
    p = create paper;
    (def title p => t)
    } puis cldbup (add_authors p xs);
def cldbup (h : t) => (h : t);
def cldbup [ ] => [ ];
```

The function *add_authors* recursively processes all the author records at the front of the list before returning the remainder of the list. The logic for processing a single author record is:

- construct a list of authors with the specified surname and initials
- if the list is empty create a new author, define their surname and initials, and bind them to the pattern "a" else bind the pattern to the head of the list of authors
- if the list of authors for the specified paper is currently undefined, then define a list with the author "a" as its single member. Otherwise add the author "a" to the existing list.

```
add_authors : paper [paper_record] -> [paper_record];
def add_authors p ((AUTHOR_REC s i) : xs) => {
    auth_list = [ a1 | for a1 in rev_surname s;
                                    initials a1 == i ];
    a = if auth_list == [ ] then
           {a2 = create author;(def surname a2 => s);
                       (def initials a2 => i) } puis a2
```

```
          else head auth_list fi;
     e_let authors = by p in
           if authors == @ then (def by p => [a])
           else e_let new_authors = append authors [a]
                in (def by p => new_authors)
           fi
     } puis add_authors p xs;
def add_authors p (h : t) => (h : t);
def add_authors p [ ] =>[ ];
```

The file of papers is then processed lazily by evaluating the following expression:

```
cldbup (read_file paper_record "Tristarp_papers");
```

7 Scoping Referential Transparency

A defining property of languages which are purely functional is said to be *referential transparency* [Hud89]. It is this property which allows functional languages to be declarative rather than imperative in nature. Put simply, referential transparency means that every instance of a particular variable has the same value within its scope of definition. This is the interpretation of variables in algebra and logic which allows the use of equational reasoning.

Although *Relief* is not a pure functional language it is possible to identify regions of referential transparency where the techniques used in functional programming, and by functional compilers, can still be applied. This follows from the fact that the concept of referential transparency is scoped by definition. In a purely functional language the scope is the environment or script in which an expression is evaluated. Referential transparency in *Relief* is scoped by the update operations which destroy reference. These regions of referential transparency can be statically identified by means of an *effects checker* which classifies the effect of an expression. Furthermore an effects checker is of use in concurrent systems since it identifies potential interference between expressions.

7.1 Gifford and Lucassen's Effects Checker

Effects checking was first described by Gifford and Lucassen [GL86] in connection with their research into the integration of imperative and functional languages. Gifford and Lucassen identify the following three kinds of effects of expressions:

- A – allocation and initialisation of mutable storage, for example the declaration of a variable or collection.
- R – reading a value from mutable storage.
- W – writing to mutable storage, that is assignment.

There are eight possible combinations of these three effects which form a lattice under the subset relation. These eight different combinations can be grouped into effect classes depending on the language requirement. Gifford and Lucassen were interested in identifying those parts of a program which could be memoised and were

amenable to concurrent evaluation. Hence they produced an effect checking system which classified expressions into one of the following four effect classes:

{ W } { W,R } { W,A} { W,R,A }	- PROCEDURE class
{ R } { R, A }	- OBSERVER class
{ A }	- FUNCTION class
{ }	- PURE class

There is a total ordering of these effect classes in the sense that they form a hierarchy of effect properties. The effect properties of the four classes can be summarised as:

- PROCEDURE – expressions of this sublanguage can define variables, read mutable storage and write mutable values.
- OBSERVER – expressions of this sublanguage can define variables and read variables. They cannot perform assignments and cannot make use of expressions with the effect class PROCEDURE.
- FUNCTION – expressions of this sublanguage can define variables but they cannot read variables or change their value. FUNCTION expressions cannot make use of PROCEDURE or OBSERVER expressions.
- PURE – a pure function has no effects and cannot be affected by the evaluation of other expressions. It can neither define variables, nor read, nor write them. A PURE expression cannot make use of PROCEDURE, OBSERVER or FUNCTION expressions.

The only expressions which can interfere with concurrency are those in the effect class PROCEDURE and these can only interfere with other expressions in the PROCEDURE and OBSERVER effect classes. Thus the ability to identify expressions of these classes would be important in order to maximise concurrency in a multi-user system. In contrast PURE, FUNCTION and OBSERVER expressions do not interfere with each other and can be run concurrently. PURE expressions can be memoised and OBSERVER expressions can be memoised between updates. We now look at how effects checking is applied in *Relief.*

7.2 The Effects Checker in *Relief*

We begin by classifying the effect properties of constructs in *Relief. Relief* has no updateable variables in the usual sense but it does have extensible and updateable types, namely user-defined functions and non-lexical entities. Thus the Gifford and Lucassen concept of the allocation of mutable storage is equivalent to the declaration of such types. With regard to the initialisation of storage, a function declared in *Relief* has a default definition so that initially its application gives only the undefined value. The extent of a non-lexical type, accessed by the *All* metafunction, is initially the empty set. Type declarations however are commands and are not referentially transparent because they alter the database schema. Since they contain no

subexpressions and cannot appear as subexpressions they are omitted from the effects checking.

With regard to updates, user-defined functions are updated by the *def* and *del* functions which define and delete equations in the definition of a function. The extents of non-lexical types are altered by the *create* and *destroy* functions. An expression which directly or indirectly calls any of these four functions has the (W) property. An expression which directly or indirectly applies a user-defined function or examines the extent of a non-lexical type has a value which is dependent on the state of the system. Such an expression has the (R) property. Since no expressions in *Relief* can have the (A) property it is sufficient to classify them as belonging to one of the following three effect classes:

{W } { W,R }	- PROCEDURE class
{ R }	- OBSERVER class
{ }	- PURE class

Because *Relief* is a single-user sequential system we can say that an expression is referentially transparent if it makes no change to mutable storage under any condition. Thus an expression with the effect class OBSERVER or PURE is referentially transparent and has a value which is independent of the evaluation order of its subexpressions. The operational semantics of *Relief* allows the static scoping of referential transparency. Thus we arrive at the definition for the scope of referential transparency for an expression in *Relief*:

"The scope of an expression's referential transparency in Relief is the maximal enclosing expression which has no (W) effects."

In a multi-user system, PURE and OBSERVER expressions can still be treated as referentially transparent if there is a locking mechanism to prevent expressions with effect class PROCEDURE from changing the function and entity types on which they depend.

Our approach differs from that of Gifford and Lucassen in that we regard the effect class of an expression as a property orthogonal to its type. We regard their approach of annotating each function type with its effect class as unnecessarily complex and giving no practical benefit. In our implementation an expression's effect class is inferred from its subexpressions in tandem with its type. Thus whilst performed in parallel effects checking and type checking are conceptually independent.

In *Relief* the programmer declares the effect class of a function from the outset and the compiler ensures that the declaration is not violated. The effect class of a user-defined function must be either PROCEDURE or OBSERVER since it is an observer of its own mutable definition and hence possesses the (R) property. A user-defined function defaults to the effect class of OBSERVER, whereas an effect class of PROCEDURE is specified by preceding the function name with "proc". The following examples illustrate this. Firstly declaring a PROCEDURE whose application conditionally creates an author and defines their surname and initials:

```
proc create_author : string string -> update;
def create_author sur init =>
    if [ a | for a in rev_surname sur;
                              initials a == init ] == [ ]
```

```
    then { a = create author; (def surname a => sur);
                        (def initials a => init) } fi;
```

Secondly declaring an OBSERVER function which is dependent on other functions. This example defines a person's salary as the sum of their basic pay and their bonus.

```
salary : person -> integer;
def salary x => basic x + bonus x;
```

Finally declaring an OBSERVER function which is only dependent on its own definition. This example extracts the second component of a tuple.

```
sec_comp : (alpha1 ** alpha2) -> alpha2;
def sec_comp (x,y) => y;
```

The effects checker ensures that the declarations are not violated by checking that the effect class of each defining equation is less than or equal to the declared effect class of the function. The inference rules used in the effects checker are given by the pseudocode below. The effect class of a pattern is PURE and since this is the least value it can be ignored where some other expression is present.

The fundamental inference rule is the one for application. The effect class of an application is the maximum of the effect class of the function and the effect class of its argument. This means that effect inference is pessimistic since it assumes that if an expression contains a side-effect then that side-effect will take place. This is of course not necessarily true with lazy evaluation. We now give the algorithm for the function *effect_chk* which infers the effect class of an expression. The declared effect class of a user-defined function is retrieved by calling the function *dec_effect*. Thus the value of "dec_effect(create_author)" is PROCEDURE assuming the declaration of create_author above.

```
effect_chk(E)   // effects checker
{
switch(E)
    case (E1 E2) // application
    case ( e_let p = E2 in E1 ) //eager let expression
    case ( l_let p = E2 in E1 ) //lazy let expression
          e1 = effect_chk E1
          e2 = effect_chk E2
          return (max(e1, e2) )
    case (\p. E)       // lambda abstraction
          return ( effect_chk E)
    case Qual     //Qualifier in a list comprehension
          case ( for p in E)      // generator
          case E                  // filter
                return ( effect_chk E)
    case VAR v  // a bound formal parameter, not
                // updateable, hence PURE
    case NONLEX n     // a nonlexical entity
    case CONFUN c     // a data constructor
    case ETYPE        // a type as a parameter
```

```
case constant c   // eg integer or string
      return (PURE)
case FUN f        // a persistent function
      return(dec_effect(f))
case INFUN f            // a built-in function
      switch(f)
      case def f a1 .. an => rhs
            e = effect_chk (rhs)
            if e > dec_effect(f) then FAIL
            return(PROCEDURE)
      case del f a1 .. an
      case create t
      case destroy t
            return(PROCEDURE)
      OTHERWISE
            return(PURE)
```

This algorithm can be modified to prohibit subexpressions with the effect class PROCEDURE from parts of the language. We suggest that this restriction might be applied to list comprehension qualifiers so that optimisations such as those published by Trinder [Tri91] can be automatically performed.

8 Related Work

Ghelli et al. [GOPT92] have proposed an extension to list comprehension notation which provides update facilities for an object-oriented database language. The extension is achieved by adding a *do* qualifier which can perform actions on objects identified by earlier bindings in the list comprehension. They show that known optimisation techniques can be modified to operate in the presence of do qualifiers. Their technique however relies on the programmer restricting the use of side-effecting expressions to do qualifiers, there being no compile-time checking.

We briefly discuss three other approaches which promise efficient, functionally pure database update: linear types, monads and mutable abstract data types (MADTs). They are all efficient since they can guarantee a single-threaded store so that updates can be done in-place.

8.1 Linear Types

A linear object is an unshared unaliased singly-referenced object and thus linear objects have the property that there is only one access path to them at any given time. A linear variable is a 'use-once' variable which can only be bound to a linear object and must be dynamically referenced exactly once within its scope. Linear type checking systems have been designed which can statically check that linear variables are neither duplicated nor discarded. These systems have a theoretical foundation in linear logic whence they derive their name.

The motivation for having linear types in functional languages is to capture the notion of a resource that cannot or should not be shared, hence the linear variable. A linear type system does however allow for the explicit creation, copying and destruction of linear objects. Whenever a program creates a linear object it must also be responsible for its destruction. One advantage of this is that there is no need for garbage collection in a purely linear language. For functional programmers linear types offer a way, amongst other things, of guaranteeing safe in-place updates since a linear object can have only a single reference.

Purely linear languages however are very restrictive since functions must always pass-on their linear arguments, explicitly destroy them or return them even when they are not changed. The programs in these languages tend to be somewhat difficult to read due to their heavy use of multiple returned values. Another problem with languages which support linear types is that they must also deal with nonlinear, i.e. shared, objects. This requires them to have a dual type checking system whereby most of the type inference rules have linear and nonlinear counterparts. The programmer must be aware of the distinction between linear and nonlinear values.

In the database context we want to treat database objects as linear when updating them but also to treat the same objects as shared when querying. However an object cannot be both linear and nonlinear at the same time. An interesting application of linear types to functional databases has been implemented by Sutton and Small [SS95] in their further development of PFL, a functional database language which uses a form of relation, called a selector, as its bulk data type. In the first version of PFL the incremental update of relations was achieved by means of the side-effecting functions *include* and *exclude*. A linear type checking system was introduced into PFL later on by changing these functions so that they took a linear relation name as an argument and returned the same relation name as a result. The type checking system worked by preventing the duplication or discarding of these relation names. Sutton and Small give examples of strict updating functions which show that the language is still update-complete within the constraints of this type checking. In order to reduce the number of parameters returned when querying they also allow a relation to be referred to by a nonlinear name. However a relation's linear name and nonlinear name cannot both appear within the same expression.

8.2 Monads

In-place update can also be guaranteed by wrapping up the state in a higher-order type which is accessed only by functions with the single-threaded property. The functions must support the creation, access and updating of the state within the type. Wadler has shown how monads can be used for this purpose [Wad95]. Monads, a category-theoretic notion, are defined by a triple: the type constructor *M*, and the operations *unit* and *bind*. Monadic programming is an implicit environment based approach to programming actions such as updating state and input/output. Actions on the environment have a special type which indicates what happens when the action is performed. The programmer composes actions with the two combinators unit and bind.

The monadic approach to handling state has both advantages and disadvantages when compared with the use of linear types. Its chief advantage is that it does not require a special type system beyond the usual Hindley-Milner one. Also because monads handle state implicitly it is argued that this can make an underlying algorithm more apparent since there is less parameter passing. Their main disadvantage is that they require a centralised definition of state. Although this restriction might initially appear to fit nicely with the use of a single conceptual data model in databases, in practice however this lack of granularity would have performance and concurrency implications. A treatment of multiple stores with different types, such as relations in a relational database, requires the definition of many monads and monad morphisms in order to support operations involving the different types. It may also be that for databases a two state monad is more appropriate in order to represent the current state and the state at the start of the transaction. This could then support the rollback operation. We consider these to be areas for further research.

Monads can also be defined in terms of *unit*, *map* and *join* operations which are a generalisation of the same operations which give the semantics of list comprehensions. Wadler has shown how the list comprehension notation can also be used to manipulate state monads [Wad90], each qualifier becoming an action on the state. The resulting programs however look very procedural.

8.3 Mutable Abstract Data Types

Monads in themselves do not enforce linearity on the encapsulated state, their operations must be carefully designed in order to ensure this. The use of abstract data types which encapsulate state and linear access to it has been investigated by Hudak [CH97]. A *mutable abstract datatype* or MADT is any ADT whose rewrite semantics permit “destructive re-use” of one or more of its arguments while still retaining confluence in a general rewrite system. A MADT is an ADT whose axiomatisation possesses certain linearity properties. Hudak shows how a number of MADTs can be automatically derived from such an ADT. Using an array as an example Hudak gives three ways of defining an appropriate MADT. These correspond to direct-style semantics, continuation passing style (CPS) and monadic semantics. A MADT is defined in terms of functions which generate, mutate and select state of a simple type.

Not only is the state hidden in a MADT but the linearity is too and so there is no need for the programs using MADTs to have a linear type system. It has been noted that programming in MADTs tends towards a continuation passing style no matter which form of MADT is used. MADTs are excellent at handling simple state but further research is required into handling bulk data types efficiently and into combining MADTs. Unlike monads there is no rigorous way of reasoning with a number of MADTs.

9 Conclusions

Whilst there is much promising research into handling state within functional languages the update problem in functional databases such as FDL remains

problemmatical because the database is comprised of function definitions which must be both simple to apply and simple to update. The design of an uncomplicated integrated language to handle both queries and updates in an efficient purely functional manner is difficult. Whilst linear types could have been used it is our view that the result would have been less satisfactory and less programmer-friendly than the approach we have taken. The introduction of the four update functions described in section 4, whilst not functionally pure, has kept the language strongly data-model oriented. Also none of the approaches outlined in the section on related work have been used to address transaction handling which is a fundamental distinction between database programming and persistent programming. In contrast *Relief* does have a programmable transaction handling mechanism although this is not described here. (See [Mer98] for details of this).

Relief demonstrates that a functional interpreter based on graph reduction can be used to perform updates and that effects checking can scope referential transparency so that the usual functional optimisations can be used for queries and read-only subexpressions. The approach taken has allowed the benefits of lazy evaluation and static type checking to be retained. Updates are performed efficiently and the programmer does not have to learn a new type system. Whilst the eager let helps to make the sequencing of updates more explicit it is still necessary to understand the operational semantics of the pattern matcher when function arguments contain side-effects. Applicative order reduction would simplify these semantics but the lazy stream model used for file reading and the other advantages of lazy evaluation would be lost. Update by side-effect and effects checking are applicable to database systems supporting other data models.

Further work will centre on integrating *Relief* with the data model of *Hydra* [AK94] which distinguishes between navigable functions and non-navigable functions in a functional database. This may require the definition of further effect classes.

Acknowledgements: We would like to thank the referees and Alex Poulovassilis for their comments during the preparation of this paper. The first author was supported by the EPSRC and IBM UK Labs Hursley during this work.

10 References

[AK94] R. Ayres and P.J.H. King: Extending the semantic power of functional database query languages with associational facilities. In Actes du Xieme Congres INFORSID, pp301-320, Aix-en-Provence, France, May 1994.

[CH97] Chih-Ping Chen and Paul Hudak: Rolling Your Own Mutable ADT – A connection between Linear Types and Monads. ACM Symposium on Principles of Programming Languages, January 1997

[Der89] M. Derakhshan: A Development of the Grid File for the Storage of Binary Relations Ph.D. Thesis, Birkbeck College, University of London, 1989

[FH88] Anthony J. Field, Peter G. Harrison: Functional Programming. Addison-Wesley 1988

[GL86] David K. Gifford and John M. Lucassen: Integrating Functional and Imperative Programming. In Proceedings of the ACM Conference on Lisp and Functional Programming, Cambridge, Massachussets, pp28-39, ACM, 1986

[GOPT92] Giorgio Ghelli, Renzo Orsini, Alvaro Pereira Paz, Phil Trinder: Design of an Integrated Query and Manipulation Notation for Database Languages, Technical Report FIDE/92/41, University of Glasgow, UK, 1992.

[Hud89] Paul Hudak: Conception, Evolution, and Application of Functional Programming Languages. ACM Computing Surveys, Vol. 21, No. 3, September 1989 pp359-411

[KDPS90] Peter King, Mir Derakhshan, Alexandra Poulovassilis, Carol Small: TriStarp - An Investigation into the Implementation and Exploitation of Binary Relational Storage Structures. Proceedings of the 8th BNCOD, York 1990

[Mer98] P.F.Meredith: Extending a Lazy Functional Database Language with Updates. Thesis for Submission. Birkbeck College, University of London, 1998

[NHS84] J. Nievergelt, H. Hinterberger, K.C. Sevcik: The Grid File: An Adaptable, Symmetric, Multikey File Structure. ACM TODS, Vol 9, No 1, March 1984, pp38-71

[Pou88] A. Poulovassillis: FDL: An Integration of the Functional Data Model and the Functional Computational Model, Proceedings of the 6th BNCOD, July 1988, Cambridge University Press, pp215-236.

[Pou92] A. Poulovassilis: The Implementation of FDL, a Functional Database Language. The Computer Journal, Vol 35, No 2, 1992

[PK90] A. Poulovassilis and P. J. H. King: Extending the Functional Data Model to Computational Completeness. Proceedings of EDBT'90, pp75-91, Venice 1990. Springer-Verlag, LNCS 416.

[Shi81] David W. Shipman: The Functionanal Data Model and the Data Language DAPLEX. ACM TODS, Vol 6, No 1, March 1981, pp140-173

[SS95] David Sutton, Carol Small: Extending Functional Database Languages to Update Completeness. Proceedings of 13th BNCOD, Manchester, 1995

[Tri91] Phil Trinder: Comprehensions, a Query Notation for DBPLs. The 3rd International Workshop on DBPLs, "Bulk Types and Persistent Data". August 1991, Nafplion, Greece. Morgan Kaufman Publishers.

[VV82] G.M.A. Verheijen, J. Van Bekkum: NIAM: An Information Analysis Method. In "Information Systems Design Methodologies: A Comparative Review", T.W.Olle et al. (eds), North-Holland, 1982

[Wad90] Philip Wadler: Comprehending Monads. ACM Conference on Lisp and Functional Programming, Nice, June 1990

[Wad95] Philip Wadler: Monads for Functional Programming. In "Advanced Functional Programming", Proceedings of the Bastad Spring School, May 1995, LNCS vol 925

Extending the ODMG Architecture with a Deductive Object Query Language

Norman W. Paton and Pedro R. Falcone Sampaio

Department of Computer Science, University of Manchester
Oxford Road, Manchester, M13 9PL, United Kingdom
Fax: +44 161 275 6236 Phone: +44 161 275 6124
{norm,sampaiop}@cs.man.ac.uk

Abstract. Deductive database languages have often evolved with little regard for ongoing developments in other parts of the database community. This tendency has also been prevalent in deductive object-oriented database (DOOD) research, where it is often difficult to relate proposals to the emerging standards for object-oriented or object-relational databases. This paper seeks to buck the trend by indicating how deductive languages can be integrated into the ODMG standard, and makes a proposal for a deductive component in the ODMG context. The deductive component, which is called DOQL, is designed to conform to the main principles of ODMG compliant languages, providing a powerful complementary mechanism for querying, view definition and application development in ODMG databases.

1 Introduction

To date, deductive database systems have not been a commercial success. The hypothesis behind this paper is that this lack of success has been encouraged by the tendency from developers of deductive databases to build systems that are isolated from the widely accepted database development environments. The fact that deductive database systems tend to be complete, free-standing systems, makes their exploitation by existing database users problematic:

1. Existing data is not there. Existing databases in relational or object-oriented systems cannot be accessed by deductive databases without transferring the data into the deductive database system.
2. Existing tools are not there. Moving to a new database system means learning a new toolset, and coping with the fact that few deductive database systems have benefited from the levels of investment that have been directed towards leading relational or object-oriented products.
3. Existing programs are not there. Software that has been developed for non deductive databases has no straightforward porting route to a deductive environment.
4. Existing experience is not useful. There are very few developers for whom a deductive database would be a familiar environment.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 149–164, 1998.

These difficulties that face any developer considering using deductive databases with existing applications mean that even where there is an application that could benefit from a deductive approach, such an approach may well not be exploited. With new applications, points (2) and (4) remain as a disincentive, and the decision rests on the widely rehearsed debate on the relative strengths and weaknesses of deductive and other database paradigms. Here we note only that the decision to use a deductive database seems to be made rather rarely. In practice, the limited range and availability of deductive database systems will often lead to an alternative approach being used by default.

In the light of the above, we believe that it is not generally viable to make a case for deductive database systems as a replacement for existing approaches. There are sure to be a number of domains for which a deductive approach is very appropriate, but it is difficult to envisage widespread uptake of 'pure' deductive database systems.

The argument that deductive languages can provide a mechanism for querying, recursive view definition and application development that is *complementary* to existing database languages is much easier to make. Where deductive features are made available within an existing database system, the user can elect to exploit the deductive mechanism or not, depending on the needs of the application. Where the deductive mechanism is closely integrated with other language facilities, multi-paradigm development is supported, and thus the deductive language can be used in niche parts of an application if more widespread exploitation is not felt to be appropriate.

The argument that deductive database languages can be complementary to those of other paradigms is not new [17], but it is not only the principle of integration, but also the practice that counts. For example, [16] describe the integration of the deductive relational database Coral with C++, but the impedance mismatches that result are extreme, and the approach can be considered more of a coupling than an integration. In ROCK & ROLL [2] the integration is much more seamless, but neither the languages nor the data model used predated the development of the ROCK & ROLL system.

The argument for deductive languages in any form can be made on the basis of application experience [10], or on the grounds that an alternative style and more powerful declarative language extends the options available to programmers. Although the case that recursion is an important motivation for including deduction in databases has been used previously to the point of tedium, we note that ODMG languages do not currently provide any clean way of describing recursive relationships. As a simple example, the implementation of the simple ancestor relationship as a method using the ODMG C++ binding is presented in figure 1, for the database in figure 3. The code would be more complex if cycle detection was required.

In what follows this paper considers *how* a deductive language can be integrated into the ODMG architecture. The decision space that relates to the incorporation of deductive facilities into the ODMG model is presented in section 2. The deductive language DOQL is described in section 3, and its integration

```
d_Set<d_Ref< Person >> * Person::ancestors const ()
{    d_Set<d_Ref<Person>> *the_ancestors = new d_Set<d_Ref<Person>>;
     //---- ancestors = parents Union ancestors (parents)
     if (father != 0) {
        the_ancestors->insert_element(father);
        d_Set<d_Ref<Person>> *f_grand_parents = father->ancestors();
        the_ancestors->union_with(*f_grand_parents);
        delete (f_grand_parents);}
      if (mother != 0)
      {  d_Set<d_Ref<Person>> *m_grand_parents = mother->ancestors();
         the_ancestors->insert_element(mother);
         the_ancestors->union_with(*m_grand_parents);
         delete (m_grand_parents);}
      return the_ancestors;
}
```

Fig. 1. C++ implementation of *ancestors()* method

into the ODMG environment is discussed in section 4. How DOQL relates to other languages for object-oriented databases is outlined in section 5, and a summary is presented in section 6. Throughout the paper, it is assumed that readers have some familiarity with the ODMG standard, and with deductive database technologies, as described in [4] and [5], respectively.

2 The Design Space

This section considers the issues that must be addressed when seeking to add deductive facilities into the ODMG standard. The fundamental questions are *what deductive facilities should be supported* and *what relationship should the deductive facilities have to other components in an ODMG database*.

2.1 The Architecture

The introduction of deductive facilities into the ODMG architecture is made straightforward by the presence of OQL as a role model. The inclusion of the deductive language in a way that mirrors that of OQL minimizes the amount of infrastructure that existing OQL users must become acquainted with to exploit the deductive language. The following architectural principles can be followed in the development of the deductive extension.

1. Existing components should not be modified, and extensions should be minimal.
2. The deductive component should make minimal assumptions about the facilities of the ODMG system with which it is being used, and should be amenable to porting between ODMG compliant systems.

The consequences of the first of these principles can be considered in the context of the existing ODMG components as follows:

- *The data model and ODL:* the inclusion of a deductive language should necessitate no changes to the structural data model, but will require some extensions to operation definition facilities, as deductive rules do not have pre-defined input and output parameters.
- *OQL:* the presence of an existing declarative language raises the question as to whether or not one should be able to make calls to the other. There is no doubt that the ability to make calls from OQL to the deductive language (and potentially the reverse) could be useful, but such extensions would impact on principle (2) above.
- *Language bindings:* the existing language bindings need not be changed to accommodate a new embedded language. The deductive language should be callable from C++, Java and Smalltalk in a manner that is reminiscent of the embedding of OQL, through the addition of new library calls for this purpose.

The consequences of the second of the principles listed above depend on whether or not the developer has access to the source code of an ODMG compliant system. The assumption that is being made (because it is true for us) is that the deductive language is being implemented without access to the source of any ODMG compliant system. This has the following consequences for the implementation:

- *Query language optimization/evaluation:* this has to be written from scratch, as reusing the existing infrastructure developed for OQL is only an option for vendors. Implementing the optimiser and evaluator specifically for the deductive language also eases porting of the deductive component.
- *Data model interface:* it is necessary to have access to a low level API for the underlying database (such as O_2API), as the deductive system often needs to perform operations on objects for which the types are not known at system compile time.

The architecture adopted for DOQL is illustrated in figure 2. The DOQL compiler and evaluator is a class library that stores and accesses rules from the ODMG database – the rule base is itself represented as database objects. The class library is linked with application programs and the interactive DOQL interface. The interactive interface is itself a form of application program.

2.2 The Language

Research on deductive languages for object databases has been underway for around 10 years now, and a considerable number of proposals have been made (see [15] for a survey covering 15 proposals). These proposals differ significantly in the range of facilities supported by the logic language, so that a spectrum can be identified in which ROLL [2] is perhaps the least powerful and ROL [12] perhaps the most comprehensive. In practice, the range of facilities supported

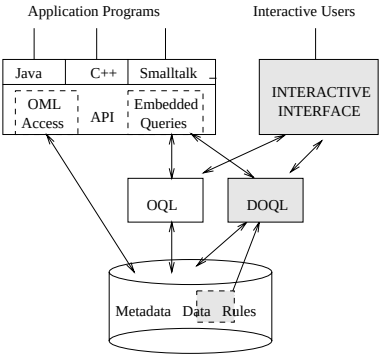


Fig. 2. Architecture diagram showing location of DOQL compiler/evaluator.

depends to a significant extent on the context. For example, ROLL is deliberately left simple because it is closely linked to the imperative language ROCK, whereas ROL is a single-language system with more limited access to complementary, non-deductive facilities. It is thus important to identify some principles that can be used to guide the development of an ODMG compliant DOOD. We suggest that an ODMG compliant deductive language should:

1. Be able to access all ODMG data types and should only be able to construct values that conform to valid ODMG data types.
2. Be usable to define methods that support overloading, overriding and late binding.
3. Be able to express any query that can be written in OQL.
4. Be usable embedded or free standing.

It is straightforward to think up additional principles (e.g. the deductive language should be usable to express integrity constraints; the deductive language should be able to update the underlying database; the deductive language should extend the syntax of OQL), but the more are added, the more constraints/burdens are placed on developers in terms of design and implementation. DOQL supports the 4 principles enumerated above, but none of those stated in this paragraph.

In general terms, the principles stated above indicate that the deductive language should be tightly integrated with the model that is at the heart of the ODMG standard, should support object-oriented programs as well as data, should extend what is already provided in terms of declarative languages, and should not be bound tightly to a specific user environment. The principles also avoid a commitment to areas of deductive language design that raise open issues in terms of semantics or the provision of efficient implementations. The language described in section 3 contains only language constructs that have been supported elsewhere before, and thus can be seen as a mainstream deductive

database language. The novelty of the approach is thus not in the individual language constructs, but in the context within which it fits.

3 The DOQL Language

This section describes the DOQL language, enumerating its constructs and illustrating them using an example application.

3.1 Example Application

The example application, from [8], describes employees and their roles within projects and companies. Figure 3 presents the structure of the database using UML [14].

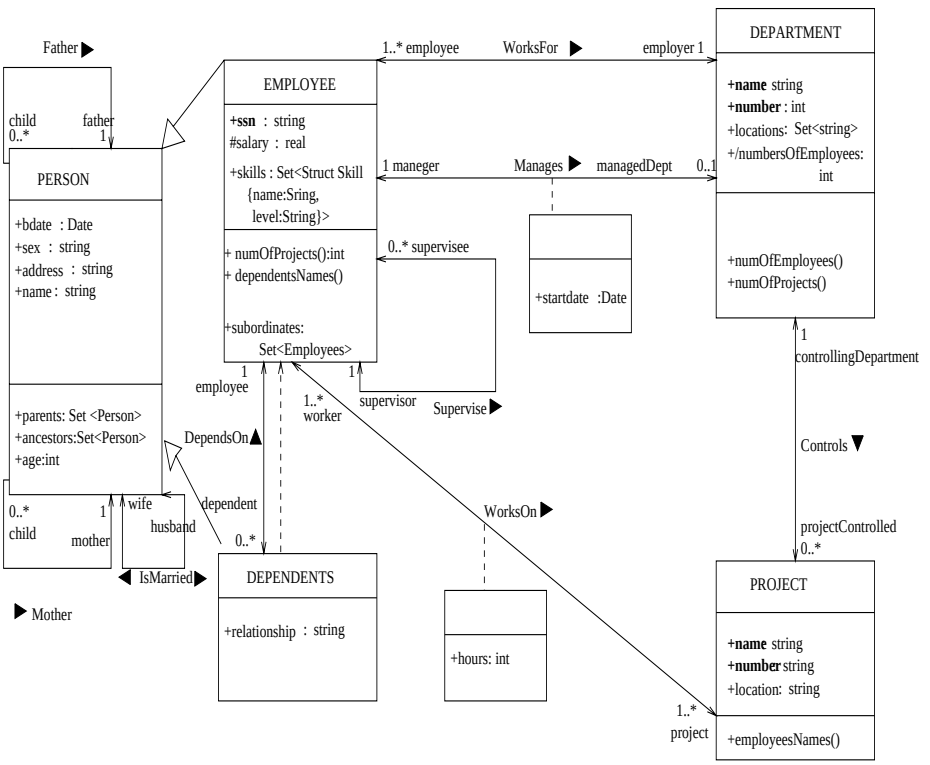


Fig. 3. Example Application

3.2 Queries

Two different approaches were considered in the definition of the syntax of the deductive language:

- Extend OQL to support recursion.
- Define a rule-based query language tailored to provide reasoning over the ODMG data model.

The main attraction of the first approach is in the possibility of reusing the existing OQL syntax. This advantage, though, does not supplant the difficulties involved in providing comprehensive deductive definitions in the straightjacket of the **Select From Where** syntax block[1]. DOQL thus adopts a syntax that is familiar from other DOOD proposals, but adapted for use in the ODMG model.

DOQL is a typed logic-based rule language supporting declarative queries and declarative rules for intensional definitions. The rule language follows the Datalog rule style, but combines the positional and named approaches to attribute denotation and extends Datalog's atoms and terms to deal with the properties of the underlying ODMG model. Variables range over instances of types declared in an ODMG schema. The language is composed of two main syntactic categories: *terms* and *literals*. Non negated literals are also called *atoms*.

Terms denote elements in the domain of interpretation and define the alphabet of the deductive language. Terms can be variables, atomic or complex literals and evaluable expressions:

- *Variables:* represented by alphanumeric sequences of characters beginning with an uppercase letter (e.g. `X`, `Names`).
- *Atomic Constants:* representing ODMG atomic literals (e.g., `2.32`, `'Alan Turing'`, `13`, `true`, `nil`).
- *Compound Constants:* representing structured or collection literals defined in the ODMG model. (e.g., `struct(name:'programming',level: 'high')`, `set(1,2,3,5)`, `bag(1,1,3,3)`, `list(1,1,2,4)`).
- *DB entry names:* representing the set of names associated with root objects and extents in an ODMG schema.
- *Expressions:* evaluable functions that return an ODMG structural element (literal or object). Each expression has a type derived from the structure of the query expression, the schema type declarations or the types of named objects. Expressions can have other terms as parameters and are divided in terms of the following categories:
  path expressions: traversal paths starting with a variable or a DB entry name, that can span over attributes, operations and relationships defined in type signatures of an ODMG schema (e.g.,`Z.age`, `john.subordinates`). Path expressions traversing single valued features in type signatures can be composed forming paths of arbitrary lengths (e.g., `john.dept.name`).

[1] It has recently been proposed that the inclusion of non-linear recursion in SQL be delayed until SQL4 [9].

aggregates and quantified expressions: predefined functions applied to terms denoting collections. Complex expressions can be formed by applying one of the operators (`avg`, `sum`, `min`, `max`, `count`, `exists`, `for-all`) to pairs *(T:F)*, where T is a term and F is a formula formed by a conjunction of literals. T and F can share common variables, and other variables can appear in F. Every variable that appears in T must also appear in F and cannot appear elsewhere outside the scope of the operator. The following expressions show the use of aggregates and quantifiers:

```
count(X.dependents), max(X.age: persons(X))
exists(X: persons(X), X.age > 110)
for-all(X: employees(X), X.age >= 18)
```

arithmetic expressions: (A + B), (A * B), (A / C), (A - C).
string concatenation: (A || B).
set expressions: (A `union` B), (A `intersect` B), (A `except` B).

Literals denote atomic formulas or negated atomic formulas that define propositions about individuals in the database. The types of atomic formulas are:

– *collection formulas*: enable ranging over elements of collections. A collection can be a *virtual collection* defined by a previous rule or a *stored collection* defined in the extensional database. Molecular groupings of collection properties can be done using the operator (`[ ]`). The operator (`=>`) can be used to finish path expressions that end in a collection, resulting in collection formulas. The following formulae show the diversity of possibilities in ranging over collections, depending on the needs of each rule. In the first formula, the identities of the collection instances represent the focus point. In the second formula, the items needed are the values for the properties `name` and `age` of each collection instance. The third formula binds variables to identities and properties of the elements in the collection. The fourth formula denotes all the subordinates of `john` and the last formula ranges over a collection of structured literals that represent information about `john`'s skills. Collections of literals can only use the molecular notation due to the absence of object identity in their elements.

```
persons(X)
persons[name=N, age=Y]
persons(X)[name=N, age=Y]
john.subordinates=>X
john.skills=>[name=N, level=L]
```

– *element formulas*: enable ranging over (1:1) relationships and object-valued attributes (without inverses) in the extensional database. The following formulae are examples of element formulas. In the first formula, `john` is a name (DB entry point) in the schema.

```
john.spouse->X
L.spouse->X
```

– *operation formulas*: denote formulas of type <*Term*> Operator <*Term*> or just <*Term*>, where the latter is a boolean expression. Operation formulas encompass relations between elements and collections (`in`), strings (`substr`, `like`, `lower`), numbers (<=, >=, <, >) and also the notions of value equality (=, <>) and Oid equality (==, !=).

```
X.age <= 90
exists(X: employees(X), struct(name:'coding',level:'high') in X.skills)
john.father == mary.father
```

– *method calls*: denoted by the syntactic form:

$< method\_name >(arg_1, ...arg_n)[- >< result >]$
$< method\_name >(arg_1, ...arg_n)[=>< result >]$
$< method\_name >(arg_1, ...arg_n)[=< result >]$

This allows user-defined ODMG operations to be invoked from DOQL. The three forms return individual elements (objects or structured literals), collections, and simple values, respectively. The invoked operations should be side-effect free, a criterion that cannot be guaranteed or tested for in existing ODMG environments. It is thus up to the user to exercise caution in embedding method invocations, and the system should support an option of forbidding method calls from within DOQL (for example, through an environment variable). It is not possible to call methods in which a single argument position is used for both input and output, as this would require assignment to a bound DOQL variable.

– *DOQL goals*: these are described below, along with consideration of DOQL rule and method definitions.

3.3 DOQL Methods and Rules

Rules are used to implement deductive methods and define virtual collections over ODMG databases. A rule is a clause of the form: $H \leftarrow L_1, L_2, L_3, \ldots L_n$, where H is the *Head* of the rule and the body is a sequence of literals denoting a formula. The syntactic form of the head depends on whether the rule is a *regular clause* or a *method clause*.

Regular Clauses In regular clauses, the head of the rule is of the form:

$< rulename > (arg_1, ..., arg_n)$

where each arg_i is a variable, an atomic constant, a compound constant, or a grouping expression applied to a term. For example, in the following rule, `D` is `local` if it is a department that controls only local projects:

```
local(D) :- departments(D),
            for-all(Projects: D.projectControlled=>Projects,
                       Projects.location='local').
```

Rules can be recursive, and can be invoked from queries or rules:

```
parent(X,Y) :- persons(X), X.father->Y.
parent(X,Y) :- persons(X), X.mother->Y.
relative(X,Y) :- parent(X,Y).
relative(X,Y) :- parent(X,Z), relative(Z,Y).
```

Grouping is the process of grouping elements into a collection by defining properties that must be satisfied by the elements. Grouping is restricted to rules that contain only a single clause and to a single argument position in the head of the clause. The grouping construct is expressed by the symbols { Var_Name } for set groupings, < Var_Name > to group elements as bags and [Var_Name | $a_i : o_i, \ldots a_n : o_n$] to group as lists, sorting the list according to the attributes $a_i, \ldots, a_n$, each attribute defining a letter (a=ascending, d=descending) for the ordering field o_i that defines the order of the elements of the list according to that attribute. The following example shows the grouping of the relatives of a person as a list sorted in ascending order of the age attribute.

```
relatives_of(X,[Y|age:a]) :- relative(X,Y).
```

Method Clauses In method clauses, the head of the rule is of the form:

$< recipient >::< method - name > (arg_1, ..., arg_n)$

where *recipient* is a variable, and arg_i are as for regular clauses. For example, the following method rule reinterprets `relative` as a deductive method on `person`:

```
extend interface PERSON {
    P::relative(R) :- parent(P,R).
    P::relative(R) :- parent(P,Z), Z::relative(R).
}
```

The approach to overriding and late binding follows that of ROCK & ROLL [7]. In essence, methods can be overridden, and the definition that is used is the most specialised one that is defined for the object that is fulfilling the role of the message recipient.

3.4 Stratification, Safety and Restrictions

DOQL adopts conventional restrictions on rule construction to guarantee that queries have finite and deterministic results. In particular: each variable appearing in a rule head must also appear in a positive literal in the rule body; each variable occurring as argument of a built-in predicate or a user-defined ODMG operation must also occur in an ordinary predicate in the same rule body or must be bound by an equality (or a sequence of equalities) to a variable of such an ordinary predicate or to a constant; all rules are stratified; a rule head can have at most one grouping expression; the type of each argument of an overridden DOQL method must be the same as the type of the corresponding argument in the overriding method; all rules are statically type checked, using a type inference system.

4 DOQL In Context

This section describes how DOQL fits into the ODMG environment, showing how rule bases are created and managed, how rules are invoked, and how DOQL is incorporated into imperative language bindings.

4.1 Managing Rule Bases

Rules can be created either transiently, or in rule bases that are stored persistently in the ODMG compliant database. Rule bases are created using the `create_rulebase` operation, which takes as parameter the name of the rulebase to be created. The inverse of `create_rulebase` is `delete_rulebase`. These commands can be run from the operating system command line, from the interactive DOQL system, or from executing programs.

Once a rulebase has been created, rules can be added to it using the `extend rulebase` command, which can also be run from the operating system command line, from the interactive DOQL system, or from executing programs. For example, figure 4 extends the rulebase `employment` with both regular and method rules.

```
extend rulebase employment {
// Extend EMPLOYEE with an operation related_subordinate
// that associates the EMPLOYEE with subordinates who are related to them
 extend interface EMPLOYEE {
    Emp::related_subordinate(R) :- Emp::relative(R), Emp::subordinate(R).
    Emp::subordinate(S) :- Emp[supervisee=>S].
    Emp::subordinate(S) :- Emp[supervisee=>Int], Int::subordinate(S).
   }
 // A dodgy manager is one who is the boss of a relative
 dodgy_manager(M) :- employees(M), exists(S: M::related_subordinate(S)).
}
```

Fig. 4. Extending a rule base.

4.2 Rule Invocation

Queries are evaluated over the current state of the database and rule base, with different query formats depending on the nature of the query. Rules can only be invoked directly from within DOQL queries and rules. The different formats are:

– Single element queries (`SELECT ANY`): this form of query returns a single element from the set of elements that results from the evaluation of the goal. The choice of the element to be returned is non-deterministic.

- Set of elements queries (`SELECT`): this form of query returns all elements that satisfy the goal.
- Boolean queries (`VERIFY`): this form of query has a boolean type as query result. The result is the value yielded by the boolean formula given as a parameter.

The query formats can be summarized according to the following general expressions:

```
SELECT [ANY] Result_Specifier
    FROM    DOQL Query
    [WITH   Rule_Statements]
    [USING  Rulebase]

VERIFY Boolean_Formula
    [WITH   Rule_Statements]
    [USING  Rulebase]
```

The `Result_Specifier` is a comma separated list of arguments that are the same as the arguments allowed in a rule head, and `Rulebase` is a comma separated list of persistent rulebase names. For example, the following query associates each employee with a set of their related subordinates, using rules from the `employment` rulebase.

```
SELECT E,{S}
FROM   E::related_subordinate(S)
USING  employment
```

The verification format requires that the operand of the verify return a boolean value:

```
VERIFY exists(E:employees(E), dodgy_manager(E))
USING  employment
```

The `WITH` clause is used to allow the specification of rules that exist solely for the purpose of answering the query (i.e. which are not to be stored in the persistent rule base). For example, the following query retrieves the names of the employees who have more than 5 related subordinates:

```
SELECT Name
FROM   employees(E)[name=Name],
       num_related_subordinates(E,Number), Number > 5
WITH   num_related_subordinates(E,count({S})) :- E::related_subordinate(S).
USING  employment
```

The above query forms can be used either in the interactive DOQL interface or in embedded DOQL.

4.3 Embedded DOQL

DOQL programs can be embedded in host programming languages in a manner similar to that used for OQL [4]. DOQL statements are embedded as parameters of calls to the API function `d_doql_execute` (d_DOQL_Query Q, Type_Res

Result)[2]. The function receives two parameters. The first parameter is a query container object formed by the query form described in section 4.2 along with any input arguments to the query. The second parameter is the program variable that will receive the result of the query.

Figure 5 is a code fragment that shows the embedded form of DOQL for C++. First some variables are declared for the input parameter to the query (`pedro`) and the result (`the_ancestors`). Then the query object `q1` is created with the query as the parameter of the constructor function. It is then indicated that `pedro` is the (first and only) parameter of `q1` – this parameter is referred to within the text of the query as `$1`. Finally, the query is executed by `d_doql_execute`, and the results are placed in `the_ancestors`.

```
d_Ref <Person> pedro = ...;
d_Set < d_Ref <Person>> *the_ancestors = new Set(Person);
d_DOQL_Query q1("SELECT Y
                 FROM relative($1,Y)
                 WITH  relative(X,Y) :- persons(X), X.father->Y.
                       relative(X,Y) :- persons(X), X.mother->Y.
                       relative(X,Y) :- relative(X,Z), relative(Z,Y).");
q1 << pedro;
d_doql_execute(q1, the_ancestors);
```

Fig. 5. Example embedded DOQL code fragment

5 Related Work

This section outlines a range of features that can be used to compare the query components of proposals that integrate declarative query languages and imperative OO programming languages in the context of OODBs.

Bidirectionality of calls: relates to the flexibility of the integration approach. In unidirectional systems, one of the languages can call the other, but not vice-versa.

Restructuring: relates to the operations available in the declarative language that can be used to reorganize the queried data elements to yield more desirable output formats.

Type system: relates to the underlying type system used across the integrated languages.

[2] A general syntax is used to show concepts. The specific syntax varies for each language binding (e.g. for C++, template<class T> void d_doql_execute(d_DOQL_Query &query, T &result)).

Query power: relates to the query classes supported by the query language. (FOLQ = first-order queries, FIXP = fixpoint queries, HOQ = higher-order queries).

View mechanism: relates to the features provided by the query language for the definition of derived attributes, derived classes through filtering, derived relations through composition, and derived classes through composition applying OID invention.

Table 1 shows a comparison between the query facilities supported by DOQL and the query components of Chimera [6], Coral++ [16], OQL [4], OQLC++ [3], Noodle [13] and ROCK & ROLL [2].

Language	**Criteria**				
	Bidirection of Calls	Restructuring Operators	Type System	Query Power	View Mechanism
Chimera	no	unnesting	Chimera object model	FOLQ, FIXP	attributes, classes (filtering), relations
Coral++	no	set grouping, unnesting	C++	FOLQ, FIXP	relations
DOQL	yes	(set,bag,list) grouping, unnesting	ODMG object model	FOLQ, FIXP	classes (filtering), relations
Noodle	yes	(set,bag) grouping, unnesting	Sword object model	FOLQ, FIXP HOQ	relations
OQL	yes	(set,bag,list) grouping, unnesting	ODMG object model	FOLQ	classes (filtering), relations, classes (OID invention)
OQLC++	yes	unnesting	C++	FOLQ	no
ROCK & ROLL	yes	unnesting	Semantic object model	FOLQ, FIXP	no

Table 1. Query languages for object databases

Some important aspects of our design are:

- language integration is done using a standard object-model as the underlying data model of the deductive system.
- the approach to integration complies with the call level interface defined by the ODMG standard, reusing existing compiler technology for other ODMG

compliant languages and providing portable deduction within the imperative languages without the need for changes in syntax or to the underlying DBMS.
- bidirectionality of calls between the integrated languages.
- powerful aggregate and grouping operators that can be used for restructuring data in applications that require summaries, classifications and data dredging.
- updating and control features are confined within the imperative language

Views in DOQL are considered subproducts of queries, and in particular, results of queries are typically associations (complex relations). This context differs from the semantics of views proposed in [11,1] by not inventing OIDs, by not having view declarations as a part of the DB schema, and by not having methods attached to views obtained through composition.

Language integration proposals can also be compared based on the seamlessness of the language integration (see [2]) or based on the object-oriented and deductive capabilities supported (see [15]). Using the criteria of [2], DOQL can be seen to support evaluation strategy compatibility, (reasonable) type system uniformity and bidirectionality, but to lack type checker capability and syntactic consistency. The lack of type checker capability means that embedded DOQL programs can cause type errors at runtime – obtaining compile time type checking of embeddded DOQL would require changes to be made to the host language compiler. The lack of syntactic consistency is unavoidable, as DOQL can be embedded in any of a range of host languages, and thus cannot hope to have a syntax that is consistent with all of them.

6 Summary

This paper has proposed a deductive language for use in ODMG compliant database systems. The principal motivation for the development of such a language is the belief that failure to make deductive systems fit in with existing database models and systems has been a significant impediment to the widespread exploitation of deductive technologies in databases.

The language that has been proposed, DOQL, can be used both free-standing and embedded within the language bindings of the ODMG model. Rules can be defined in a flat clause base or as methods attached to object classes. The paper has described both the language and how it relates to other ODMG components both in terms of the implementation architecture and as a user language.

Acknowledgements: The second author is sponsored by Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil) – Grant 200372/96-3.

References

1. S. Abiteboul and A. Bonner. Objects and views. In *Proc. of the ACM-SIGMOD Int. Conference on Management of Data*, pages 238–247, 1991. 163
2. M. L. Barja, N. W. Paton, A. A. Fernandes, M. Howard Williams, and Andrew Dinn. An effective deductive object-oriented database through language integration. In *Proc. of the 20th VLDB Conference*, pages 463–474, 1994. 150, 152, 162, 163, 163
3. J. Blakeley. OQLC++: Extending C++ with an object query capability. In Won Kim, editor, *Modern Database Systems*, chapter 4, pages 69–88. Addison-Wesley, 1995. 162
4. R. Cattel and Douglas Barry, editors. *The Object Database Standard: ODMG 2.0.* Morgan Kaufman, 1997. 151, 160, 162
5. S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases.* Springer-Verlag, Berlin, 1990. 151
6. S. Ceri and R. Manthey. Consolidated specification of chimera (cm and cl). Technical Report IDEA.DE.2P.006.1, IDEA - ESPRIT project 6333, 1993. 162
7. A. Dinn, N. W. Paton, M. Howard Williams, A. A. Fernandes, and M. L. Barja. The implementation of a deductive query language over an OODB. In *Proc. 4th Int. Conference on Deductive and Object-Oriented Databases*, pages 143–160, 1995. 158
8. R. Elmasri and S. Navathe. *Fundamentals of Database Systems 2nd. Edition.* Addison-Wesley, 1994. 154
9. S. Finkelstein, N. Mattos, I. Mumick, and H. Pirahesh. Expressing recursive queries in sql. Technical Report X3H2-96-075r1, International Organization for Standardization, 1996. 155
10. O. Friesen, G. Gauthier-Villars, A. Lefebvre, and L. Vieille. Applications of deductive object-oriented databases using del. In Raghu Ramakrishnan, editor, *Applications of Logic Databases*, chapter 1, pages 1–22. Kluwer Academic Publishers, 1995. 150
11. W. Kim and W. Kelley. On view support in object-oriented database systems. In W. Kim, editor, *Modern Database Systems*, chapter 6, pages 108–129. Addison-Wesley, 1995. 163
12. M. Liu. Rol: A deductive object base language. *Information Systems*, 21(5):431–457, 1996. 152
13. I. S. Mumick and K. A. Ross. Noodle: A language for declarative querying in an object-oriented database. In *Proc. of the Third Intl. Conference on Deductive and Object-Oriented Databases*, volume 760 of *LNCS*, pages 360–378. Springer-Verlag, 1993. 162
14. Rational Software Corporation. *Unified Modeling Language 1.0 - Notation Guide*, 1997. 154
15. Pedro R. F. Sampaio and Norman W. Paton. Deductive object-oriented database systems: A survey. In *Proceedings of the 3rd International Workshop on Rules in Database Systems*, volume 1312 of *LNCS*, pages 1–19. Springer-Verlag, 1997. 152, 163
16. D. Srivastava, R. Ramakrishnan, P. Seshadri, and S. Sudarshan. Coral++: Adding object-orientation to a logic database language. In *Proc. of the 19th VLDB Conference, Dublin, Ireland*, pages 158–170, 1993. 150, 162
17. J. Ullman and C. Zaniolo. Deductive databases: Achievements and future directions. *ACM - SIGMOD Records*, 19(4):75–82, December 1990. 150

The Chase of Datalog Programs ⋆

Nieves R. Brisaboa[1], Agustin González[2], Héctor J. Hernández[2] ⋆⋆, and José R. Paramá[1]

[1] Facultad de Informática. Univ. da Coruña. A Coruña. Spain.
{brisaboa,parama}@udc.es
[2] Laboratory for Logic, Databases, and Advanced Programming, Dept. of Co. Science, NMSU, Las Cruces, NM, USA 88003-0001.
{hector,agonzale}@cs.nmsu.edu

Abstract. The chase of datalog programs is a new way to reason about datalog programs that are evaluated on databases consistent with a set of constraints. It is an equivalence-preserving program transformation that uncovers properties of datalog programs that must hold when they are evaluated on consistent databases. Here, we summarizes our research.

1 Introduction

To understand the interaction of datalog programs with that class of databases we define the *chase* [2,3] in the context of deductive databases. Our chase outputs programs which are, in general, "simpler" than the input programs, since they will contain fewer distinct variables as a result of the equating of variables done by the chase.

Example 1. Let $P = \{r_0, r_1\}$, where:

$$r_0 = p(X,Y) \ :- \ e(X,Y)$$
$$r_1 = p(X,Y) \ :- \ e(Z,X), e(X,Z), p(Z,Y)$$

Let $F = \{f = e : \{1\} \rightarrow \{2\}\}$; f means that if we have two atoms $e(a,c)$ and $e(a,b)$ in a database consistent with F, then $c = b$. Let us denote the databases that are consistent with F by $SAT(F)$. Then chasing in the natural way [2,3] the trees for the expansions $r_1 \circ r_0$, $r_1 \circ r_1 \circ r_0$, $r_1 \circ r_1 \circ r_1 \circ r_0$, ..., that is, equating variables, as implied by the fd f, in atoms defined on e in those expansions, we can obtain the datalog program $P' = \{s_0, s_1, s_2\}$, the chase of P wrt F, where:

$$s_0 = r_0$$
$$s_1 = p(X,X) : -e(Z,X), e(X,Z), p(Z,X)$$
$$s_2 = p(X,Z) : -e(Z,X), e(X,Z), p(Z,Z)$$

⋆ This work was partially supported by NSF grants HRD-9353271 and HRD-9628450, and by CICYT grant Tel96-1390-C02-02.

⋆⋆ H.J. Hernández is also affiliated with Inst. de Ing. y Tec., Univ. Autónoma de Cd. Juárez, Ch., México

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 165–166, 1998.

P and P' are equivalent on databases in $SAT(F)$. The idea behind the chase of a datalog program is to obtain an equivalent datalog program such that the recursive rules have a least number of different variables; variables in the original recursive rule(s) get equated accordingly to the fds given. In fact, because of the equating of variables, our chase transforms P into P', which is equivalent to a nonrecursive datalog program on databases in $SAT(F)$!

Thus the chase of datalog programs is a very powerful rewriting technique that let us uncover properties of datalog programs that are to be evaluated on consistent databases. In particular, we can use it to find a minimal program, one without redundant atoms or rules, that is equivalent to the given program over databases in $SAT(F)$. This extends Sagiv's results in [4] regarding containment, equivalence, and optimization of datalog programs that are evaluated on $SAT(F)$. We can use the chase of programs to uncover other properties, e.g., to find out whether a set of fds F implies an fd g on a program P [1]; F *implies* g on P if for any database d in $SAT(F)$, $P(d)$ satisfies g. The following example illustrates this.

Example 2. Assume that P'' consists of the rules $r_2 = p(X,Y) \ :- e(X,Y)$ and $r_3 = p(X,Y) \ :- e(X,Y), e(X,Z), p(Z,Y)$. Then it is difficult to see why $F = \{f = e : \{1\} \to \{2\}\}$ implies on P'' the fd $g = p : \{1\} \to \{2\}$. The chase of P'' wrt F helps us to see why.

Since the chase of P'' wrt F is the program $\{r_2, p(X,Z) \ :- e(X,Z), p(Z,Z)\}$, it is now easy to see that on any database the fixed point of p is exactly a copy of the facts defined on e — because the atoms defined on p and e in both rules have exactly the same arguments— and, thus, for any database d satisfying F, $P''(d)$ satisfies g. Therefore the chase of P'' allows us to see that F implies g on P''.

Using the chase of datalog programs, we prove that: (1) the FD-FD implication problem [1] is decidable for a class of linear datalog programs, provided that the fds satisfy a minimality condition; (2) the FD-FD implication problem can still be decided when we relax the linearity and minimality conditions on datalog programs of the previous class, provided that the set of recursive rules of the program preserves a set of fds derived from the input fds.

References

1. S. Abiteboul and R. Hull. Data Functions, Datalog and Negation. In *Porc. Seventh ACM SIGACT-SIGMOD-SIGART. Symposium on Principle of Database Systems,* pp 143-153, 1998. 166, 166
2. A.V. Aho, C. Beer, and J.D. Ullman. The Theory of Joins in Relational Databases. *ACM-TODS*, 4(3) pp 297-314, 1979. 165, 165
3. D. Maier, A.O. Mendelzon, and Y. Sagiv. Testing Implications of Data Dependencies. *ACM-TODS*, 4(4) pp 455-469, 1979. 165, 165
4. Y. Sagiv. Optimizing Datalog Programs. In J. Minker, editor *Foundations of Deductive Databases and Logic Programming*, chapter 17 pp 659-698. Morgan Kauffmann Publishers, 1987. 166

Describing and Querying Semistructured Data: Some Expressiveness Results

Natasha Alechina[1] and Maarten de Rijke[2]

[1] School of Computer Science, University of Birmingham
Birmingham, B15 2TT, England
N.Alechina@cs.bham.ac.uk
http://www.cs.bham.ac.uk/~nxa
[2] ILLC, University of Amsterdam, Pl. Muidergracht 24
1018 TV Amsterdam, The Netherlands
mdr@wins.uva.nl
http://www.wins.uva.nl/~mdr

Data in traditional relational and object-oriented databases is highly structured and subject to explicit schemas. Lots of data, for example on the world-wide web is only *semistructured*. There may be some regularities, but not all data need adhere to it, and the format itself may be subject to frequent change.

The important issues in the area of semistructured data are: how to describe (or constrain) semistructured data, and how to query it. It is generally agreed that the appropriate data model for semistructured data is an edge-labeled graph, but beyond that there are many competing proposals. Various constraint languages and query languages have been proposed, but what is lacking so far are 'sound theoretical foundations, possibly a logic in the style of relational calculus. So, there is a need for more works on calculi for semistructured data and algebraizations of these calculi' [Abiteboul 1997].

One of the main methodological points of this paper is the following. There are many areas in computer science and beyond in which describing and reasoning about finite graphs is a key issue. There exists a large body of work in areas such as feature structures (see, for example, [Rounds 1996]) or process algebra [Baeten, Weijland 1990,Milner 1989] which can be usefully applied in database theory. In particular, many results from modal logic are relevant here. The aim of the present contribution is to map new languages for semistructured data to well-studied formal languages and by doing so characterise their complexity and expressive power.

Using the above strategy, we study several languages proposed to express information about the format of semistructured data, namely data guides [Goldman, Widom 1997], graph schemas [Buneman et al. 1997] and some classes of path constraints [Abiteboul, Vianu 1997]. Among the results we have obtained are the following:

Theorem 1. *Every set of (graph) databases defined by a data guide is definable by an existential first-order formula.*

Theorem 2. *Every set of (graph) databases conforming to an acyclic graph schema is definable by a universal formula.*

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 167–168, 1998.

Every set of (graph) databases conforming to an arbitrary graph schema is definable by a countable set of universal formulas from the restricted fragment of first-order logic.

We also argue that first-order logic with transitive closure FO(TC) introduced in [Immerman 1987] is a logical formalism which is best suited to model navigational query languages such as Lorel [Abiteboul et al. 1997] and UnQL [Buneman et al. 1997]. We give a translation from a Lorel-like language into FO(TC) and show that the image under translation is strictly less expressive than full first-order logic with binary transitive closure.

Theorem 3. *Every static navigational query is expressible in FO(TC).*

Theorem 4. *The image under translation into FO(TC) of static navigational queries is strictly weaker than FO(TC) with a single ternary predicate Edge and binary transitive closure.*

In our ongoing work, we are investigating the use of FO(TC) for query optimisation, and we are determining complexity characterisations of the relevant fragments of FO(TC), building on recent results by [Immerman, Vardi 1997] on the use of FO(TC) for model checking.

References

Abiteboul 1997. Abiteboul, S.: Querying semi-structured data. Proc. ICDT'97 (1997) 167

Abiteboul et al. 1997. Abiteboul, S., Quass, D., McHugh, J., Widom, J., Wiener, J.L.: The Lorel query language for semistructured data. J. of Digital Libraries, **1** (1997) 68–88 168

Abiteboul, Vianu 1997. Abiteboul, S., Vianu, V.: Regular path queries with constraints. Proc. PODS'97 (1997) 167

Baeten, Weijland 1990. Baeten, J.C.M., Weijland, W.P.: Process Algebra. Tracts in Theoretical Computer Science, Vol. 18. Cambridge University Press (1990) 167

Buneman et al. 1997. Buneman, P., Davidson, S., Fernandez, M., Suciu, D.: Adding structure to unstructured data. Proc. ICDT'97 (1997) 167, 168

Goldman, Widom 1997. Goldman, R., Widom, J.: DataGuides: enabling query formulation and optimization in semistructured databases. Proc. VLDB'97 (1997)

Immerman 1987. Immerman, N.: Languages that capture complexity classes. SIAM J. of Comput. **16** (1987) 760 – 778 168

Immerman, Vardi 1997. Immerman, N., Vardi, M.: Model Checking and Transitive Closure Logic. Proc. CAV'97 (1997) 291–302 168

Milner 1989. Milner, R.: Communication and Concurrency. Prentic Hall (1989) 167

Rounds 1996. Rounds, W.C.: Feature logics. In: van Benthem, J., ter Meulen, A. (eds.): Handbook of Logic and Language. Elsevier (1996) 167

The TENTACLE Database System as a Web Server

Marc Welz[1] and Peter Wood[2]

[1] University of Cape Town
Rondebosch, 7701 RSA
mwelz@cs.uct.ac.za
[2] King's College London
Strand, London, WC2R 2LS UK
ptw@dcs.kcl.ac.uk

Abstract. We describe the TENTACLE system, an extensible database system which uses a graph data model to store and manipulate poorly structured data. To enable the database to be tailored to particular problem domains, it has been equipped with a small embedded interpreter which can be used to construct the equivalent of customised views of or front-ends to a given semi-structured data domain. To demonstrate the capabilities of the system we have built a web server directly on top of the database system, making it possible to provide a clean mapping between the logical structure of the web pages and the underlying storage system.

1 Introduction

TENTACLE is an extensible database system which attempts to take a different approach to modelling, manipulating and presenting complex or poorly structured application domains. As data model we have chosen a graph-based approach. Graph models provide a flexible environment in which complex associations can be represented naturally without an intermediate translation phase from the domain to the storage abstraction. Our model is similar to the OEM used by LORE [2].

The TENTACLE data manipulation subsystem consists of an integrated imperative programming and declarative query language. The language runs directly inside the database system and enables the user to program the database server. This removes the need for gateway programs and wrappers as encountered in more typical database systems. In this respect our system bears some resemblance to database programming languages [4].

We have chosen the world-wide web as an example application to illustrate how the above-mentioned features of the database system can be applied to the task of supporting complex and or poorly structured problem domains.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 169–170, 1998.

2 Implementation

The database is implemented as a monolithic system, where the integrated query and host language interpreter is coupled to the storage subsystem so that the entire database runs as a single process, reducing the inter-component communications overhead.

The TENTACLE database system has been written from the ground up. Its lowest layer is a specialised graph storage system which maps the graph components onto disk blocks. The services provided by this layer are used by the graph management system to provide basic graph creation and manipulation services. On top of the graph management layer we have built a re-entrant parser capable of concurrently interpreting several programs written in the combined programming and query language.

3 Application

The world wide web is the largest networked hypertext system. Surprisingly its interaction with typical databases has so far been quite limited—almost all hypertext documents are stored on a conventional file system and not in a database.

In the cases where databases are indeed accessible via the world wide web, the interaction between the web and the database is not direct—in almost all cases such "web-enabled" databases are simply regular databases accessible through a web-based gateway (see CGI [1]). Such an arrangement may not only cause performance problems (a result of the interprocess communications overheads of web server, gateway(s) and database), but also result in impedance mismatches, since usually the database and web use different data models (relational for the database, network for the web server).

Our system attempts to address these issues by moving the web server into the TENTACLE database system. For each incoming connection, TENTACLE activates a program fragment stored in the database itself which provides the web server functionality (by implementing parts of the HTTP [3] protocol). This program uses the incoming request (URL) to traverse the database graph and sends matching nodes or materialised information to the client. Since the mapping from the graph structure to the world wide web is reasonably direct, there is no need to perform cumbersome or costly rewriting operations.

References

1. Common gateway interface (CGI) specifications. http://hoohoo.ncsa.uiuc.edu/cgi/. 170
2. S. Abiteboul, D. Quass, J. McHugh, J. Widom, and J. L. Wiener. The LOREL query language for semistructured data. *International Journal on Digital Libraries*, 1(1):68–88, April 1997. 169
3. T. Berners-Lee, R. Fielding, and F. H. Hypertext transfer protocol HTTP/1.0. http://ds.internic.net/rfc/rfc1945.txt, May 1996. 170
4. S. M. Clamen. Data persistence in programming languages: A survey. Technical Report 155, School of Computer Science, Carnegie Mellon University, May 1991. 169

A Toolkit to Facilitate the Querying and Integration of Tabular Data From Semistructured Documents.

L.E. Hodge, W.A. Gray and N.J.Fiddian
Department of Computer Science
University of Wales, Cardiff, UK
e-mail: {scmleh|wag|njf}@cs.cf.ac.uk

Abstract. The advent of the World Wide Web (WWW) has meant there is a growing need to link information held in databases and files with information held in other types of structure [2]. This poster presents a toolkit which enables information held in tables within documents to be linked with data held in conventional databases. The toolkit addresses the problems of:-

- extracting tabular data from various types of document e.g. semi-structured text [4], HTML and spreadsheets
- providing a standard interface to these tables via a standard query language
- extracting useful table meta-data from the table and accompanying text.

Tables are a very effective way of presenting information. Because of their flexibility and ease of reading, tables have long been used as a method of presenting information in a clear and concise manner. With the introduction of the Internet, large bases of unstructured textual information have become available, many of which contain information in a tabular form, an example being results from scientific studies appearing within a paper. There is a growing awareness of this information and the need to link it with more conventionally held information in databases if the full potential of this information is to be realized.

The problem with tabular information is that although the nature of its presentation makes it easy for the reader to understand, it is not possible to access and utilize this information automatically in a computer application and so link it with other information. Part of our research is to produce a set of tools that will facilitate the generation of wrappers [1,3] for diverse tables embedded in various types of document. These wrappers will enable access to the information contained within the tables via a query language such as SQL and so enable them to be linked with other information, particularly that held in relational databases.

When designing and creating a table to represent a set of information, a table designer may need to modify the information in order to present it in the most useful way. In

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 171-172, 1998.

performing these transformations the designer uses knowledge about the information to alter the way that it is represented. This can be as simple as introducing a 'total' column by adding two columns or converting a percentage mark into a grade. In other cases, complex formulae may be applied to derive the results required for display. Although the designer finally has the table in the form that he wishes it to be displayed it would be useful for us to know how the source information has been manipulated to produce the information displayed in the table. In some cases such information is available in the tables meta-structure (e.g. in a spreadsheet), where as in others it is buried in accompanying text.

As well as developing tools to provide access to information contained within tables, we are developing tools that can extract meta-data from the tables and any surrounding text in a document. This meta-data will essentially consist of any formulae and constraints relating to entries in the table and any information about the representation being used. The problem is that this kind of useful information is usually lost or unavailable. Some types of tables are more useful than others because they store formulae and constraints as part of their definition (e.g. spreadsheets), whereas in other representations of tables such information is simply discarded. Fortunately, information describing how columns or rows of the table are calculated can often be found in the text accompanying the table.

In addition to the tools allowing access to tables embedded within textual documents, we are developing tools to allow access to tables in the form of spreadsheets. Spreadsheets are probably the most common source of tabular data and have a high level of structure. When dealing with spreadsheets, meta-data such as formulae and constraints are easy to locate because this type of information is embedded within the spreadsheet itself. When it comes to textual documents we will need to use techniques based on NLP and data mining to locate and extract the relevant formulae and constraints from the accompanying text – a much more challenging task.

References

[1] Wrapper Generation for Semi-structured Internet Sources. Ashish, N & Knoblock, C. University of Southern California.

[2] Learning to Extract Text-based Information from the World Wide Web. Soderland, S. University of Washington. In Proceedings of Third International Conference on Knowledge Discovery and Data mining (KDD-97).

[3] Wrapper Induction for Information Extraction. Kushmerick, N, Weld, D, Doorenbos, R. Proceedings of IJCAI 97.

[4] Using Natural Language Processing for Identifying and Interpreting Tables in Plain Text. Douglas. S, Hurst. M, Quinn, D. December 1994.

Parallel Sub-collection Join Algorithm for High Performance Object-Oriented Databases

David Taniar° J. Wenny Rahayu+

° Monash University - GSCIT, Churchill, Vic 3842, Australia
+ La Trobe University, Dept. of Computer Sc. & Comp. Eng., Bundoora, Vic 3083, Australia

1 Introduction

In *Object-Oriented Databases* (OODB), although path expression between classes may exist, it is sometimes necessary to perform an explicit join between two or more classes due to the absence of pointer connections or the need for value matching between objects. Furthermore, since objects are not in a normal form, an attribute of a class may have a collection as a domain. Collection attributes are often mistakenly considered merely as set-valued attributes. As the matter of fact, *set* is just one type of collections. There are other types of collection. The Object Database Standard *ODMG* (Cattell, 1994) defines different kinds of collections: particularly *set*, *list/array*, and *bag*. Consequently, object-oriented join queries may also be based on attributes of any collection type. Such join queries are called *collection join queries* (Taniar and Rahayu, 1996).

Our previous work reported in Taniar and Rahayu (1996, 1998a) classify three different types of collection join queries, namely: *collection-equi join*, *collection-intersect join*, and *sub-collection join*. In this paper, we would like to focus on sub-collection join queries. We are particularly interested in formulating a parallel algorithm based on the sort/merge technique for processing such queries. The algorithms are non-trivial to parallel object-oriented database systems, since most conventional join algorithms (e.g. hybrid hash join, sort-merge join) deal with single-valued attributes and hence most of the time they are not capable of handling collection join queries without complicated tricks, such as using a loop-division (repeated division operator).

Sub-collection join queries are queries in which the join predicates involve two collection attributes from two different classes, and the predicates check for whether one attribute is a sub-collection of the other attribute. The sub-collection predicates can be in a form of *subset*, *sublist*, *proper subset*, or *proper sublist*. The difference between proper and non-proper is that the proper predicates require both join operands to be properly sub-collection. That means that if both operands are the same, they do not satisfy the predicate. The difference between subset and sublist is originated from the basic different between sets and lists (Cattell, 1994). In other words, subset predicates are applied to sets/bags, whereas sublists are applied to lists/arrays.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 173-174, 1998.

2 Parallel Sort-Merge Join Algorithm for Sub-collection Join

Parallel join algorithms are normally decomposed into two steps: *data partitioning* and *local join*. The partitioning strategy for the parallel sort-merge sub-collection join query algorithm is based on the *Divide and Partial Broadcast* technique.

The Divide and Partial Broadcast algorithm proceeds in two steps. The first step is a *divide* step, where objects from both classes are divided into a number of partitions. Partitioning of the first class (say class *A*) is based on the first element of the collection (if it is a list/array), or the smallest element (if it is a set/bag). Partitioning the second class (say class *B*) is exactly the opposite of the first partitioning, that is the partitioning is now based on the last element (lists/arrays) or the largest element (sets/bags).

The second step is the *broadcast* step. In this step, for each partition *i* (where *i*=1 to *n*) partition *Ai* is broadcasted to partitions *Bi* .. *Bn*. In regard to the load of each partition, the load of the last processor may be the heaviest, as it receives a full copy of *A* and a portion of *B*. The load goes down as class *A* is divided into smaller size (e.g., processor 1). Load balanced can be achieved by applying the same algorithm to each partition but with a reverse role of *A* and *B*; that is, divide *B* based on the first/smallest value and partition *A* based on the last/largest value in the collection.

After data partitioning is completed, each processor has its own data. The join operation can then be carried out independently. The local joining process is made of a simple sort-merge and a nested-loop structure. The sort operator is applied to each collection, and then a nested-loop construct is used in joining the objects through a merge operator. The algorithm uses a nested-loop structure, because of not only its simplicity but also the need for all-round comparisons among all objects.

In the merging process, the original join predicates are transformed into predicate functions designed especially for collection join predicates. Predicate functions are the kernel of the join algorithm. Predicate functions are boolean functions which perform the predicate checking of the two collection attributes of a join query. The join algorithms use the predicate functions to process all collections of the two classes to join through a nested-loop. Since the predicate functions are implemented by a merge operator, it becomes necessary to sort the collections. This is done prior to the nested-loop in order to avoid repeating the sorting operation.

References

Cattell, R.G.G. (ed.), *The Object Database Standard: ODMG-93*, Release 1.1, Morgan Kaufmann, 1994.

Taniar, D., and Rahayu, W., "Object-Oriented Collection Join Queries", *Proceedings of the International Conference on Technology of Object-Oriented Languages and Systems TOOLS Pacific'96 Conference*, Melbourne, pp. 115-125, 1996.

Taniar, D. and Rahayu, J.W., "A Taxonomy for Object-Oriented Queries", a book chapter in *Current Trends in Database Technology*", Idea Group Publishing, 1998a (in press).

Taniar, D. and Rahayu, J.W., "Parallel Collection-Equi Join Algorithms for Object-Oriented Databases", *Proceedings of International Database Engineering and Applications Symposium IDEAS'98*, IEEE Computer Society Press, Cardiff, UK, July 1998b (to appear).

Component DBMS Architecture for Nomadic Computing

J.A. McCann, J.S. Crane

High Performance Extensible Systems Group
Computer Science, City University, London, UK
{jam, jsc}@cs.city.ac.uk

Abstract. Current database management system technology is unable to provide adequate support for Nomadic Computing. Mobile applications require systems which are sufficiently lightweight and customisable to provide high performance while consuming minimal power, yet extensible enough to adapt to a constantly changing environment. Current DBMS architectures do not provide this level of customization or adaptability. In this paper we introduce Component-Based Database Management Systems (CBDMS) and discuss their suitability for mobile computing.

1. DBMS Architectures are Obsolete?

New technology is placing greater expectations on DBMS. That is, we anticipate greater use of wide-area networks, heterogeneous platforms, information sharing, higher flexibility and dynamic system modification such as 'plug and play' and hot swap. Historically, enhancing features were just added to the DBMS kernel thickening it and reducing performance. This solution will not work for mobile DBMS applications as the mobile unit is very limited in resources .

Industry, has identified a number of key application areas which would benefit from mobile DBMS. These are: healthcare, sales force automation, tourism and transportation. In particular, sales force automation allows the salesperson to use a unit connected through a mobile network to a set of databases. Furthermore, they can quickly configure complex customer solutions and customised financing options *while with the customer*. This type of operation requires that the mobile client can *update* the DBMSs as the transaction occurs which is still *not* being addressed by the DBMS community.

Performance and adaptability are key requirements of mobile data processing environments which existent DBMS architectures fail to provide. These present new challenges in areas such as query processing, data distribution and transaction management which are being re-thought in terms of performance, battery life and less reliable communication systems.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 175-176, 1998.

2. A new adaptive DBMS architecture based on Components

To provide the amount of flexibility to enable mobile data processing, we require a combination of a lightweight yet extensible DBMS. We look to the history of operating systems for an answer. Early monolithic and micro-kernel operating systems were limited in extensibility and performance, so research looked at extensible kernels, and more recently component-based operating systems [Kostkova96], as a solution.

Current DBMS technology is monolithic and is neither lightweight nor flexible enough to support mobile computing. There has been attempts to make DBMS architectures more lightweight, or to be extensible, [Boncz94], however they all are unsuitable for general DBMS applications. Therefore, DBMS need to be componentised where only the components required for a particular operation are loaded, saving performance and power costs. Furthermore, a component DBMS can extend its functionality on demand -- it binds its components at run-time, adapting to new environments. For example, the movement from a wireless network to a fixed network can trigger a new resource manager or query optimiser to be dynamically loaded as the system is no longer constrained by battery power and wireless communications. Furthermore, as processing is componentised, its migration between mobile network cells becomes more fluid.

3. Our Work and Conclusion

In this paper we have shown that to balance performance efficiency, power efficiency and cost efficiency in nomadic computing, we need more than new transaction processing and query optimisation strategies. For a DBMS to operate efficiently in a mobile environment it needs to be lightweight and customisable, providing high performance while consuming minimal power. Furthermore, the architecture must adapt to a constantly changing environment. We believe that an alternative DBMS architecture based on components is a viable solution. To this end we are currently implementing a CDBMS, which is already beginning to demonstrate its level of real-time configurability and improvement in performance, for not only mobile applications but more traditional applications.

4. References

Boncz P., Kersten M.L., ’Monet: an impressionist sketch of an advanced database system’, BIWIT'95-. Basque Int. Workshop on Information Technology Spain, July 1995.

Kostkova P., Murray K., Wilkinson T.: Component-based Operating System, 2nd Symposium on Operating Systems Design and Implementation (Seattle, Washington, USA), October 1996

ITSE Database Interoperation Toolkit

W Behrendt[1], N J Fiddian[1], W A Gray[1], A P Madurapperuma[2]

[1] Dept. of Computer Science, Cardiff University, UK
{Wernher.Behrendt, N.J.Fiddian, W.A.Gray}@cs.cf.ac.uk

[2] Dept of Computer Science, University of Colombo, Sri Lanka
ajith@cmb.ac.lk

Abstract. The ITSE system (Integrated Translation Support Environment) is a toolkit whose architecture is aimed at enabling the exchange of data between different database interoperation tools as well as the dynamic addition of new tools. Each of the tools addresses a specific set of interoperation problems and achieves interoperation using an intermediate semantic representation to map from source to target expressions.We give a brief overview of the system, the experiences that led to its specific architecture, and some scenarios in which the integrated use of tools would be beneficial. **Keywords**: Tools, Interoperation, Distributed DBMS.

System evolution requirements and tool support. Many organisations require legacy database systems to co-operate alongside modern DBMSs to support business functions. These organisations require system evolution methodologies and tools to evolve systems in their current, heterogeneous setup. The need for appropriate interoperation tools and the lack of integration between such tools is expressed by Brodie and Stonebraker [1]. Typical tasks in system evolution are the analysis of existing database schemas to ensure the conceptual models still reflect business practice; interoperation of new applications with legacy systems; the migration of these older systems to newer platforms, often coupled with a need for extensive re-design; the decoupling of remote user interfaces from monolithic, central information systems, with the added requirement of transparent access to heterogeneous information systems; or the re-interpretation of business data from different angles - one of the main drivers of the development of data warehouses.

The ITSE project (’95-’98) investigates an integration framework for a set of database interoperation tools which have diverse functionality but use a common construction paradigm by viewing interoperation as a translation task between formal languages which stand in a semantic equivalence relation to one another. The ITSE toolkit is a second-generation system providing a framework for these first-generation database interoperation tools built over the last ten years. The new system improves on previous techniques and provides facilities for data interchange between components of the toolkit. This ability is a requirement for cost-effective restructuring of information systems because it adds to the automation of otherwise error-prone design and implementation tasks for which little methodological support has been offered to date.

Architecture. The four main components of the ITSE toolkit are the *Interoperation Workbench* which houses individual interoperation tools (e.g. schema transformers, query translators, analysis and visualisation tools, etc); *Access Tools* utilizing call-level

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD’98, LNCS 1405, pp. 177-178, 1998.

interfaces to databases (including O/JDBC); the *Configurer* - a systems management tool which declares data sources to the ITSE system, and allows administrators to define which of the workbench tools are available in a specific setup; *Metatools* - a set of utilities to declare and bind new tools to the toolkit. Metatools are aimed at developer organisations whereas Configurer and Workbench are aimed at system administrators or analysts. An extensible client/server architecture allows several modes of operation for the tools: via graphical user interfaces or a scripting language, or via an agent communication language in which interoperation tasks can be requested remotely.

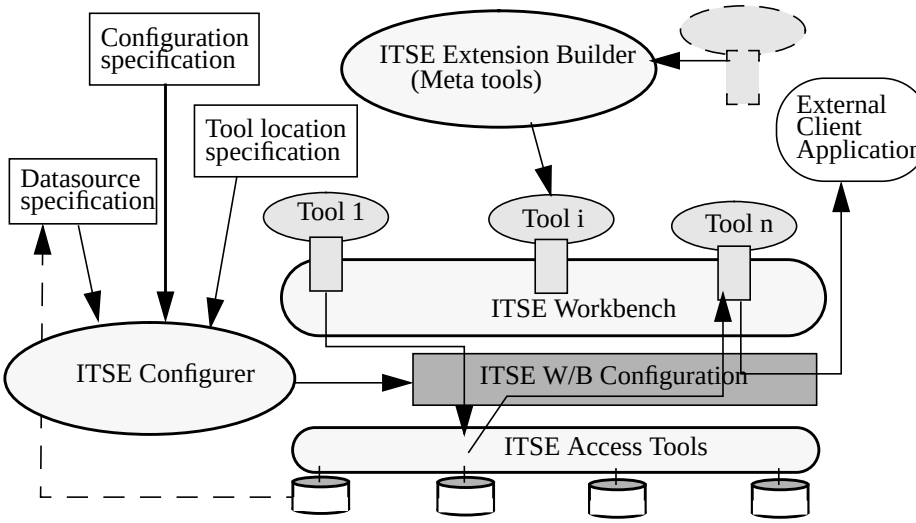


Fig.1. Top-level Functionality of the ITSE Interoperation Toolkit

Prototype implementation. ITSE server modules are implemented in BinProlog [2] with client GUIs written in Java. Connections to network-aware DBMSs are made using either ODBC and JDBC interfaces, or connectivity software developed by ourselves. The workbench includes tools for schema and query translation, as well as visualisation. Incorporating more workbench tools and connecting to different DBMSs is an ongoing process. A set of databases using Oracle, UniSQL, Versant and P/FDM forms a testbed for experimentation and development of further interoperation tools.

Ongoing and further work. The next step is to set up typical usage scenarios to test the usability and utility of the integrated tools. Examples for such scenarios could be schema analysis, on-line query translation, or MDBMS view integration. Complex scenarios include legacy system migration tasks and data warehousing. We also work on embedding Prolog-engines in C++ or Java for CORBA-based distributed services [3].

References

[1] Brodie M L, Stonebraker M, Migrating Legacy Systems, Morgan Kaufmann Publishers, 1995.

[2] ftp://clement.info.umoncton.ca/BinProlog

[3] http://www.omg.org/about/wicorba.htm

Ontological Commitments for Multiple View Cooperation in a Distributed Heterogeneous Environment

A-R. H. Tawil, W. A. Gray, and N. J. Fiddian

Department of Computer Science
University of Wales College of Cardiff, U. K.
{Abdel-Rahman.Tawil,N.J.Fiddian,W.A.Gray}@cs.cf.ac.uk

This paper deals with combining multidatabase systems with knowledge representation schemes. The current proposal is based on work done in the University of Manchester on a Medical Ontology [Rec94], database interoperation work at Cardiff University in Wales [DFG96], in addition to much related work in the area of semantic interoperability (e.g,[Wie94,ACHK93,KS94,Gru91]).

We are currently investigating a methodology for achieving interoperability among heterogeneous and autonomous data sources in a framework of knowledge sharing and re-use, based on a multi-view architecture. In this paper we describe a novel approach for integrating a set of logically heterogeneous object oriented (OO) and/or relational databases. It is based on using a view integration language capable of reconciling conflicting local classes and constructing homogenised, customised and semantically rich views. The integration language uses a set of fundamental operators that integrate local classes according to their semantic relationships and it supports *ontological commitments*, e.g., agreements to use a shared domain ontology [1] in a coherent and consistent manner. We view ontological commitments as a very important requirement in our work. In our approach, each view should commit to a particular ontology to enable other views to infer the kinds of concepts they can interpret and deal with. Here, ontological commitment specifies a commitment to the use of a term and to the interpretation of that term as described in a shared domain ontology.

The approach to our multiple-view knowledge sharing and re-use is briefly described in the following way. During the schema integration process, and specifically during the analysis phase of the integration process, the knowledge accrued fro m identifying the semantic relationships across two schema objects is encapsulated within the generated schema (view). This knowledge is costly to collect and maintain, and is hard to share and re-use. We therefore propose the organisation of this knowled ge by levels of semantic granularity (schema, tables and attribute levels), gathering semantic information at progressively more refined levels of schematic heterogeneity. Hence, during the application of each integration operator of our integration langu age, the integrator is prompted to

[1] A shared ontology is a library of reusable context independent models of the domain

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 179–180, 1998.

define/verify the explicit assumptions made in the identification of relationships among the integrated objects. These captured assumptions are represented as metadata extracted from the domain specific ontology and asso ciated with each class or attribute of the integrated schema. Also, with each operator applied, a new expression adding this knowledge is compositionally generated using the power of a compositional description logic language.

By abstracting the representational details of the views and capturing their information content, we provide the basis for extending querying facilities to enable the spanning of multiple views based upon the interpretation of the semantic knowledge which *previously* was encapsulated in each view.

The poster will show the architecture of our integration system and the structure and types of tool it supports. We will illustrate how by using domain ontologies we can enrich the database metadata with descriptive domain knowledge. We will also describe how the integration system makes use of this domain knowledge to ease the integration process and to create integrated views which maintain a *declarative*, *object oriented*, *structured* models of their domain of expertise and of the domain of expertise of each of their information sources. Thus, providing a level of understanding of the information being integrated.

References

ACHK93. Yigal Arens, Chin Y. Chee, Chun-Nan Hsu, and Craig A. Knoblock. Retrieving and integrating data from multiple information sources. *International Journal on Intelligent and Cooperative Information Systems*, 2(2):127–158, 1993. 179

DFG96. R. M. Duawiri, N. J. Fiddian, and W. A. Gray. Schema integration metaknowledge classification and reuse. In *In proceedings of the 14th British National Conference on Databases (BNCOD14), Edinburgh*, pages 1–17, UK, July 1996. 179

Gru91. T. R. Gruber. The role of common ontologies in achieving sharable, reusable knowledge bases. In E. Sandewall J. A. Allen, R. Fikes, editor, *Principles of Knowledge Representation and Reasoning: Proceedings of the Second International Conference, Cambridge, MA*, pages 601–602. Morgan Kaufmann, 1991. 179

KS94. V. Kashyap and A. Sheth. Semantic based information brokering. In *Proceedings of the 3rd Intl. Conf. on Information and Knowledge Systems*, November 94. 179

Rec94. A. Rector. Compositional models of medical concepts: Towards reusable application-independent medical terminologies. In P. Barahona and J. Christensen, editors, *Knowledge and Decisions in Health Telematics, IOS Press*, pages 133–142, 1994. 179

Wie94. Gio Wiederhold. Interoperation mediation and ontologies. *Proceeding in international symposium on fifth generation computer systems (FGCS94), Workshop on heterogeneous cooperative knowledge-bases, ICOT, Tokyo, Japan*, W3:33–48, December 1994. 179

Constraint and Data Fusion in a Distributed Information System *

Kit-ying Hui ** and Peter M. D. Gray

Department of Computing Science, King's College
University of Aberdeen, Aberdeen, Scotland, UK, AB24 3UE
{khui,pgray}@csd.abdn.ac.uk

Abstract. Constraints are commonly used to maintain data integrity and consistency in databases. This ability to store constraint knowledge, however, can also be viewed as an attachment of instructions on how a data object should be used. In other words, data objects are annotated with declarative knowledge which can be transformed and processed.
This abstract describes our work in Aberdeen on the fusion of knowledge in the form of integrity constraints in a distributed environment. We are particularly interested in the use of constraint logic programming techniques with off-the-shelf constraint solvers and distributed database queries. The objective is to construct an information system that solves application problems by combining declarative knowledge attached to data objects in a distributed environment. Unlike a conventional distributed database system where only database queries and data objects are transported, we also ship the constraints which are attached to the data. This is analogous to the use of footnotes and remarks in a product catalogue describing the restrictions on the use of specific components.

1 Constraints, Databases and Knowledge Fusion

We employ a database perspective where integrity constraints on a solution database are used to describe a design problem. This solution database can be visualised as a database storing all the results that satisfy the design problem. The process of solving the application problem, therefore, is to retrieve data objects from other databases and populate this solution database while satisfying all the integrity constraints attached to the solution database and all data objects. In practice, this solution database may not hold any actual data but provides a framework for specifying the problem solving knowledge.

To facilitate knowledge reuse, we utilise a multi-agent architecture (figure 1) with KQML-speaking software components. The user inputs the problem specifications through a user-agent in the form of constraints. These piece of knowledge, together with other constraint fragments from various resources, are sent to a constraint fusing mediator which then composes the overall description as a constraint satisfaction problem.

* This research is part of the KRAFT research project [1] funded by EPSRC and BT.
** Kit-ying Hui is supported by a grant from BT.

S.M. Embury et al. (Eds.): Advances in Databases - BNCOD'98, LNCS 1405, pp. 181–182, 1998.

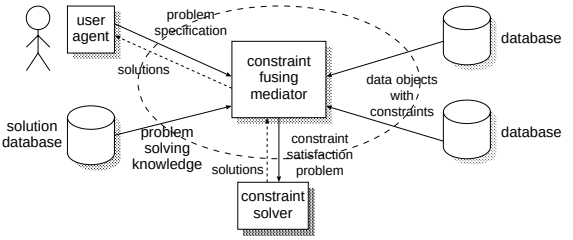


Fig. 1. *Constraint fragments are fused by the mediator to compose a constraint satisfaction problem which is solved by a constraint solver.*

2 Distributed Database Queries and Constraint Solving

The knowledge fusion process produces a declarative description of the constraint satisfaction problem (CSP) which is independent of the problem solving strategy. Given a constraint solver and a set of database servers, the choice of using constraint logic programming (CLP) techniques or generating distributed database queries depends on the complexity of the CSP, capabilities of the constraint solver and efficiency of the database servers. A constraint solver is more efficient in solving complex constraints while database queries can reduce the initial variable domains and relieve network traffic by removing incompatible solutions. A good solving strategy, therefore, should fully utilise the advantages of the two paradigms while complementing their weaknesses by each other.

3 Current Status and Future Plan

We have implemented an early prototype of the system with agents programmed either in Java or Prolog, and communicating through a subset of KQML. Constraints are expressed in the CoLan language [2] for P/FDM. Constraint transformation and fusion are based on first order logic and implemented in Prolog. At the moment we are experimenting the use of a constraint logic programming system, ECLiPSe, to solve the composed CSP. Work is still in progress to divide labour between distributed database queries and constraint solving.

References

1. P. Gray et al. *KRAFT: Knowledge fusion from distributed databases and knowledge bases.* In R. Wagner, editor, Eighth International Workshop on Database and Expert System Applications (DEXA-97), pages 682-691. IEEE Press, 1997. 181
2. N. Bassiliades and P. M. D. Gray. *CoLan: A function constraint language and its implementation.* Data & Knowledge Engineering 14 (1994) pp203-49. Also `http://www.csd.abdn.ac.uk/~pfdm`. 182

Author Index