Linköping Studies in Science and Technology
Thesis No 446

Object Views of Relational Data in Multidatabase Systems

by

Gustav Fahl

Submitted to the School of Engineering at Linköping University in partial fulfillment of the requirements for the degree of Licentiate of Philosophy

Department of Computer and Information Science
S-581 83 Linköping, Sweden

Linköping 1994

ABSTRACT

In a multidatabase system it is possible to access and update data residing in multiple databases. The databases may be distributed, heterogeneous, and autonomous. The first part of the thesis provides an overview of different kinds of multidatabase system architectures and discusses their relative merits. In particular, it presents the AMOS multidatabase system architecture which we have designed with the purpose of combining the advantages and minimizing the disadvantages of the different kinds of proposed architectures.

A central problem in multidatabase systems is that of data model heterogeneity: the fact that the participating databases may use different conceptual data models. A common way of dealing with this is to use a canonical data model (CDM). Object-oriented data models, such as the AMOS data model, have all the essential properties which make a data model suitable as the CDM. When a CDM is used, the schemas of the participating databases are mapped to equivalent schemas in the CDM. This means that the data model heterogeneity problem in AMOS is equivalent to the problem of defining an object-oriented view (or object view for short) over each participating database.

We have developed such a view mechanism for relational databases. This is the topic of the second part of the thesis. We discuss the relationship between the relational data model and the AMOS data model and show, in detail, how queries to the object view are processed.

We discuss the key issues when an object view of a relational database is created, namely: how to provide the concept of object identity in the view; how to represent relational database access in query plans; how to handle the fact that the extension of types in the view depends on the state of the relational database; and how to map relational structures to subtype/supertype hierarchies in the view.

A special focus is on query optimization.

Preface

The thesis consists of two parts. The first part gives an overview of different kinds of multidatabase system architectures and discusses their relative merits. In particular, it presents the AMOS multidatabase system architecture, which we have designed with the purpose of combining the advantages and minimizing the disadvantages of the different kinds of architectures.

A central problem in multidatabase systems is that of data model heterogeneity; the fact that the participating databases use different conceptual data models. Most systems use a *canonical data model (CDM)* to handle this. When this approach is used, the schemas of the participating databases are mapped to equivalent schemas in the CDM. These schemas can be seen as *views* of the underlying databases.

A view is a logical description of data which is derived from some other logical description of data. Views are sometimes called *virtual databases*, since no data is stored in a view. All commands against a view are translated into commands against the underlying schema according to some *mapping* between the view and the underlying schema. The fact that the view is a virtual database is transparent from users. It behaves just as if it had been the conceptual schema of a physical database. There may be several layers of views, i.e. views may defined over other views.

The most common usage of views is the external schema level in the ANSI/SPARC three-level architecture. Here, the purpose of the view is to hide some information from a user group, and/or to provide them with a transformed view of data which better suits their needs. The data model of the view is always the same as the data model of the schema it is defined over.

When views are used in multidatabase systems, the data model of the view may be different from the data model of the underlying schema. The purpose of defining views in the CDM is to give users a common interface to different kinds of databases.

Traditionally, most multidatabase systems have used the relational data model as the CDM. Nowadays, it is generally agreed that the CDM should be semantically richer than the relational data model. Otherwise it may not be possible to capture all the semantics of the participating databases. The data model we use as the CDM in AMOS is a functional and object-oriented data model. We will show that it has all the essential properties which make a data model suitable as the CDM.

When an object-oriented data model is used as the CDM, the data model heterogeneity problem is equivalent to the problem of defining *object-oriented views* (or *object views* for short) over each of the participating data-

bases.

The current database market is totally dominated by relational databases, which makes *object views of relational data* particularly important. This is the topic of the second part of the thesis.

An object view of a database means that its data can be accessed as if it was stored in an object-oriented database. The term 'object-oriented database' has been used for very different things. Commercially available object-oriented databases can usually be characterized as 'persistent C++'. The query languages of these products are usually very simple, and the basic paradigm for access to data is navigational. Research prototypes of object-oriented databases have concentrated more on providing query languages that are at least as powerful as SQL (i.e. 'relationally complete'). When we use the term 'object-oriented database' in this thesis, we mean the latter type of systems, unless otherwise stated.

Contributions

The first part of the thesis gives an overview of multidatabase system architectures and terminology. It discusses the relative merits of the architecture alternatives. In a way it also serves as a 'meta-overview', since it compares existing overviews of the area and the often confusing conflicts in terminology used by different authors. We also introduce the AMOS multidatabase system architecture, which is not really a new architecture, but rather a combination of some of the existing ones.

The main contribution of the thesis is the second part, in particular the discussion on how to process and optimize queries against an object view of relational data. The relationship between the relational data model and object-oriented data models is fairly well understood, and quite a lot of work has been done on how to transform relational schemas into schemas in an object-oriented data model. However, we are not aware of any work which discusses query processing and optimization for such view mechanisms.

A normal form for representing subtype/supertype relationships in relational schemas is introduced. The use of the normal form greatly simplifies the mapping between the relational database and the object view.

The thesis shows how the concept of object identity can be provided in the object view. It presents a way to represent relational database access in query plans which makes it possible for the query optimizer of the object view mechanism to generate optimal execution strategies. It also shows how one can handle the fact that the extension of types in the object view depends on the state of the relational database.

Delimitations

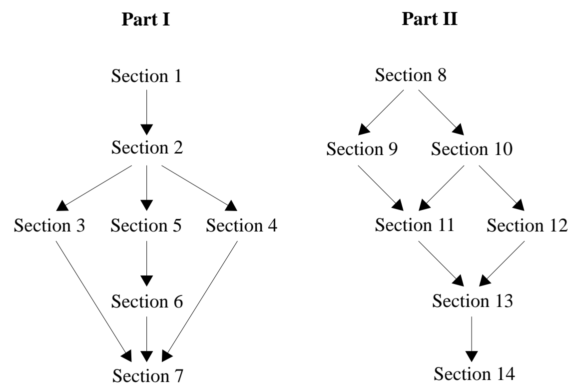
The presentation is kept informal throughout the thesis. We do not attempt to prove the equality of the semantics of the object view and the relational database. The purpose has been to present the intuition behind query

processing techniques in the object view, and we often rely on examples to do this.

We do not discuss updates to the view, only retrievals. We do not discuss optimization of recursive queries against the view. And types in the view can only have one direct supertype, i.e. we do not discuss multiple inheritance in the object view.

Thesis Overview

The dependencies between the sections of the thesis are shown in the figure below. It should be possible to read the two parts independently of each other, with one exception: The AMOS data model and query language which are used in the examples throughout the second part are introduced in sections 6.1 and 6.2. These sections should be read before part two.



Acknowledgements

I am greatly indepted to my advisor, Professor Tore Risch, for his support, for having answers to all my questions, for the constant flow of new ideas, and last but not least, for providing the AMOS platform.

I would also like to thank all members of the Laboratory for Engineering Databases and Systems (EDSLAB).

Contents

Part I Multidatabase System Architectures **1**

1 Introduction **3**

2 Overview of Architectures **5**
2.1 Main alternatives 5
2.1.1 Global Schema 7
2.1.2 Multiple Integrated Schemas 8
2.1.3 Federated 8
2.1.4 Multidatabase Language 9
2.1.5 Comparison of architectures 9
2.2 Distributed databases 11
2.3 Snapshots and materialized views 11
2.4 Multi-lingual approaches 12
2.5 No canonical data model 13
2.6 Combinations 14

3 Terminology and Standards **15**
3.1 Reference schema architecture 16

4 A Look at the Real World **19**
4.1 Legacy systems 19
4.2 Commercial state-of-the-art 20

5 Canonical Data Model **23**
5.1 Semantic modelling constructs 24
5.1.1 Essential 24
5.1.2 Complementary 25
5.2 Other aspects 26

6 The AMOS Multidatabase System **29**
6.1 Architecture 29
6.2 The AMOS data model 32
6.3 The AMOS query language 36
6.4 The AMOS data model as a CDM 39

7 Summary of Part I **41**

Part II Object Views of Relational Data **43**

8 Introduction **45**
8.1 The company example 47
8.2 Introduction to query processing 50

9 The Relational Data Model **51**

10 Object-Oriented Data Models **53**

11 Mapping Between the Relational and an Object-Oriented Data Model **55**

11.1 From object-oriented to relational 57

11.2 From relational to object-oriented 58

11.3 A normal form for representing subtype/supertype relationships in relational databases 60

12 Query Processing in Object-Oriented DBMSs **63**

12.1 Internal representation of query plans 64

12.2 Heuristic vs cost-based optimization 67

12.3 Query processing in AMOS 68

13 Object Views of Relational Data **73**

13.1 Representing relational database access in query plans 76

13.1.1 Naive strategy 76

13.1.2 Multi-way foreign functions 77

13.1.3 r-functions and r-predicates 78

13.2 Object identity 80

13.2.1 oidmap tables 81

13.2.2 oidmap functions and predicates 83

13.2.3 oidmap1 functions and predicates 86

13.2.4 Deletion semantics 90

13.3 The instance-of relationship 92

13.3.1 The typesof function 93

13.4 Query optimization 98

13.4.1 Compile-time unification 98

13.4.2 Removal of oidmap1 predicates 100

13.4.3 Substitution of r-predicates 102

13.4.4 Example 109

14 Summary of Part II **113**

Concluding Remarks **115**

Future work 115

References **117**

Index **123**

Part I Multidatabase System Architectures

1 Introduction

A multidatabase system is a system where it is possible to access and update data from multiple databases. The databases may be *distributed*, i.e. they may reside on different nodes in a computer network. They may be *heterogeneous*, i.e. different data models and query languages may be used, and even if this is not the case, they may be semantically heterogeneous.[1] And they may be *autonomous*, i.e. they may be managed independently of each other by separate organizations who wish to retain complete control over data and query processing at their site.

Multidatabase systems started to attract attention in the early 80's, when the limitations of conventional distributed database systems were realized. Conventional, homogeneous, distributed database systems are conceivable if you can build your information system from scratch. However, this is rarely possible. Most organizations have several existing databases of different kinds which are used by a large number of applications. These applications are often crucial to the organization's day-to-day work and it is very difficult, if not impossible, to replace these old systems with new technology without causing major disturbances.

And even if the information system *can* be built from scratch, you may not want to run all applications against a single, homogeneous database system. If the amount of information is large, it may be impractical to organize all data according to a single schema. And different kinds of database systems differ in their suitability for different applications. For example, a relational database may be the best choice for an administrative application, an object-oriented (OO) database may be best for an engineering application, and a hard real-time application may require some special data management. Still, it is sometimes desirable to use data from these different data sources together. They must somehow interoperate. This is the motivation for multidatabase systems.

We will use the term *component databases* when referring the databases that participate in a multidatabase system.

The rest of this part (I) of the thesis is organized as follows.

An overview of multidatabase system architectures is given in section 2. Section 3 discusses terminology and standards in the area of multidatabase

1. Semantic heterogeneity arises because there are always multiple ways to model some information, even if the same data model is used. For example, identical objects could have different names, they could be modelled using different schema constructs, there may be different levels of abstraction, different currencies could be used to represent prices, etc.

systems. Section 4 gives a short introduction to the problems of integrating existing information systems in multidatabase systems and presents state-of-the-art for commercially available multidatabase systems. Section 5 discusses the role of the canonical data model (CDM) and what properties a data model should have to be suitable as a CDM. The AMOS multidatabase system architecture, the AMOS data model, and the AMOS query language are presented in section 6. Finally, section 7 gives a summary of this part of the thesis.

2 Overview of Architectures

This section gives an overview of possible architectures for multidatabase systems. Section 2.1 presents the main alternatives. They are all based on a canonical data model (CDM) and different kinds of non-materialized views. Section 2.2 discusses the differences and similarities between multidatabase systems and conventional distributed database systems. Snapshots and materialized views are discussed in section 2.3. Multi-lingual approaches, i.e. when different languages can be used to access data, are discussed in section 2.4. Section 2.5 concerns architectures that do not use a CDM. Finally, section 2.6 discusses combinations of the 'pure' architecture alternatives.

2.1 Main alternatives

The architectures presented in this section can be regarded as the main alternatives for multidatabase system architectures. They have all in common that they use a canonical data model (CDM) to handle the problem of data model heterogeneity, and that users access the component databases through different kinds of non-materialized views. Queries and updates against these views are translated to queries and updates against the underlying views/schemas. The difference between the architectures lies in what kinds of views are provided, and in who is responsible for providing and maintaining these views.

An overview of the schema architectures of the different approaches is given in figure 1. It shows the schema architecture in a situation where users at three sites (A, B, C) access data from three data sources.[2] The terms *local schema, component schema*, and *federated schema* follow the terminology from [56], which is further discussed in section 3.[3]

---

2. The sites of the data sources need not be distinct from the sites of the users. The data sources may be local databases of the users at sites A, B, and C respectively.
Figure 1, as well as all other illustrations of schema or software architectures in this paper, shows human end-users using the different types of schemas directly (ad hoc usage). In reality, the direct user of a schema will often be an application program. Human end-users access data through the user interface of the application program. An application programming interface is provided to developers of application programs.

3. For ease of discussion, we do not use the concepts of *export schema* and *external schema* from [56] in this paper. See section 3.

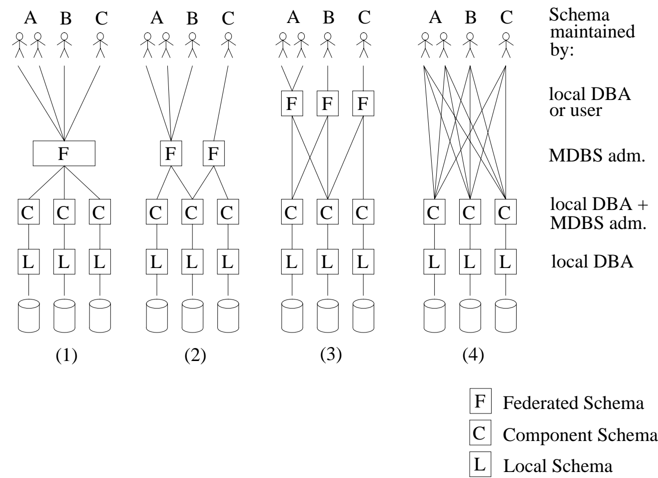


Figure 1: Schema architectures for the multidatabase system architectures discussed in section 2.1: (1) Global Schema, (2) Multiple Integrated Schemas, (3) Federated, and (4) Multidatabase Language. Users at sites A, B, and C access data from three data sources. The Local Schemas are expressed in different data models. Component Schemas and Federated Schemas are expressed using the CDM.

A *local schema* is the conceptual schema of a component database system. Since the component database systems may use different data models, the local schemas may be expressed in different data models.

For each local schema, there is a corresponding *component schema*. The component schema represents the same information as the local schema, but the CDM is used instead of the data model of the component database system. A query against a component schema is translated to queries against the underlying local schema. The results of these queries are then processed to form an answer to the initial query. All component schemas are expressed in the CDM.

A *federated schema* is an integration of multiple component schemas [6]. It makes it possible to access data from multiple databases as if it was stored in a single database. A query against a federated schema is translated to queries against the underlying component schemas. The results of these queries are then processed to form an answer to the initial query. All federated schemas are expressed in the CDM. We will use the term *integrated schema* synonymously with federated schema.

Note that the boundaries between the architectures discussed here are not completely distinct. To some extent, they can be seen as a continuum from

relatively tightly coupled systems (the global schema approach) to very loosely coupled systems (the multidatabase language approach). It is also attractive to have an architecture that is a combination of some of the approaches discussed here. Indeed, this is the case of the AMOS architecture which will be described in section 6. However, to discuss the properties of possible architectures, we have found it useful to distinguish between four different kinds of architectures.

The terms used in this paper for these approaches are (1) Global Schema, (2) Multiple Integrated Schemas, (3) Federated, and (4) Multidatabase Language. Note that the terminology used in the area of multidatabase systems is rather confused and inconsistent. For example, the term 'federated database system' has sometimes been used for a much wider class of systems than here. Section 3 gives an overview of the terminology used in other papers for the architecture alternatives discussed here.

Recall that one of the motivations for multidatabase systems was that existing applications which use a component database must continue to function without change when that database starts to interoperate with other databases. These applications will access the component databases directly through a local schema. This kind of access is not represented in figure 1.

2.1.1 Global Schema

In the *global schema* (or *single integrated schema*) approach, all component schemas are integrated into a single federated schema.[4] The federated schema is maintained by a multidatabase administration.[5] The federated schema is available to all users of the multidatabase system, and all access to the component schemas must go through the federated schema.

The global schema approach has been criticized, mainly for violating the autonomy of the component database systems. Its critics have assumed a tight coupling between the component database systems, which would make it similar to a conventional distributed database system.

A tight coupling between the component database systems is *possible* in the global schema approach, but not *necessary.* If the multidatabase system interacts with the component database systems just like any other application would, and there are no global integrity constraints, then the autonomy of the component database systems is preserved. This is done at the expense of performance and integrity. Conflicts are allowed to exist between data in the component databases, but this is somehow resolved in

4. 'Global' here means 'covering all components', not 'worldwide'.
5. The multidatabase administration plays a role similar to that of a DBA in a conventional database system. It is responsible for defining the federated schema(s) that it has agreed to provide. It must accomodate the federated schema(s) when the underlying schemas change.

the federated schema.

A more serious drawback of the global schema approach is that *everything* must be integrated. All structural and semantic conflicts between component schemas must be resolved, even for information that may never be used together. The global schema approach becomes totally infeasible when the number of component databases is very large.

The disadvantages of the global schema approach makes it unrealistic in most cases. There are however some advantages which makes it attractive for small-size systems where a tight coupling is wanted. Once the integrated schema is provided, access is simplified for users. They can work with data as if it was stored in a single database. Also, the integration solutions are shared - conflicts are resolved once and for all. And integrity can be enforced by global constraints on the integrated schema.[6]

2.1.2 Multiple Integrated Schemas

In this approach, the multidatabase administration maintains *multiple federated schemas*. Different parts of the component schemas are integrated into different federated schemas. Users access the component databases through one of the federated schemas. One federated schema is provided for each user group, i.e. for each type of integration needs. A federated schema may be shared by users at different sites.

A federated schema here provides an integrated view of those parts of the component schemas that are of interest to a particular user group, and nothing more. That is, integration solutions must be found only for information that is known to be used together. This is different from the global schema approach where *everything* has to be integrated. Different federated schemas can exist for the same set of component schemas.

What makes the global schema approach something more than a special case of this approach is that in the global schema approach the *possibility exists* to have a more tight coupling between component database systems. A single federated schema is a prerequisite if global integrity constraints are to be maintained.

2.1.3 Federated

The term *federated* database system was introduced by Heimbigner and McLeod in [30]. This term has later been used for a much wider class of systems than originally, sometimes as a synonym to ’multidatabase system’. In this paper, it is used solely for the kind of architecture described in [30].

The federated approach is similar to the previous approach (multiple inte-

6. At the expense of the autonomy of component database systems.

grated schemas), and the difference is more organizational than technical. A federated database system is more loosely coupled since no multidatabase administration is involved. It is the responsibility of the users (or the DBA) at each site to develop and maintain the federated schemas.

Using the terminology of [30], each participating database system provides an *export schema* which describes the data it wishes to share with others. This corresponds to what is called a *component schema* in this paper. A user who wishes to access data from foreign databases first has to define an *import schema*. The import schema is an integrated view of those parts of the different export schemas that are of interest to the user. All subsequent access to the foreign databases goes through the import schema. The import schema corresponds to what is called a *federated schema* here.

Since the federated schemas are provided by users or DBAs, rather than by a global administration, they can not be shared by users at different sites. They can, however, be shared by users at the *same* site.

2.1.4 Multidatabase Language

The multidatabase language approach is different from the previous approaches in that users do not access data through an integrated schema. They know that they are working against multiple databases (there is no mechanism which makes this transparent). Instead, the data manipulation language they use provides constructs to access and combine data from multiple databases.

Multidatabase languages were originally discussed in a context where all component databases were relational, i.e. no component schema was needed since all component databases used the same data model [43]. Some recent work discusses multidatabase languages where all component databases are object-oriented [46]. The concept of multidatabase languages can easily be extended to handle data model heterogeneity by adding the layer of component schemas (as in figure 1).

In the ’pure’ multidatabase language approach, which is the one considered here, the language does not allow users to create views over multiple databases. If the multidatabase language *does* provide view definition capabilities, the boundary between this approach and the federated approach starts blurring. It may even be argued that a (very short-lived) view is created every time a query in the multidatabase language is formed.

2.1.5 Comparison of architectures

As mentioned above, the boundaries between these architectures are not completely distinct. The global schema approach is to some extent a special case of the multiple integrated schemas approach. On the other hand, depending on the degree of autonomy and heterogeneity of the component

	(1) Global Schema	(2) Multiple Integrated Schemas	(3) Federated	(4) Multidata-base Language
Integrated schema - simpler access	3	3	3	1
Consistency can be maintained	3	1	1	1
Sharing of integration solutions	3	3	2	1
Possible for large systems	1	3	3	3
Only relevant information integrated - integration needs need not be decided in advance.	1	2	2	3
Conflicts resolved	3	3	3	1
No information is hidden by integration[a]	1	1	1	3
No global administration is needed	1	1	3	3
Flexible	1	2	2	3

a. This is related to the previous aspect - 'conflicts resolved'. By resolving conflicts between component databases, such as naming conflicts or inconsistent values, some information is lost in the integrated view. Hence, a high grade for 'conflicts resolved' gives a low grade here, and vice versa.

Table 1: Comparison of multidatabase system architectures. The highest grade is 3, the lowest grade is 1.

database systems, a global schema system may be very close to a conventional distributed database system. The difference between the federated and multiple integrated schemas approaches is more organizational than technical. And if the multidatabase language provides view definition capabilities, the federated approach begins to look like a subset of the multidatabase language approach.

A comparison of the architectures is given in table 1. We have listed nine desirable properties of multidatabase systems and estimated to what extent

the different approaches hold these properties. The highest grade given is 3, the lowest is 1. It is *not* a good idea to calculate the sum of two columns and come to the conclusion that one of the approaches is better than the other. The grades should be seen as an illustration of the strengths and weaknesses of the different approaches. A better idea is to observe that no approach has higher grades for all aspects, and come to the conclusion that which approach is the best depends on the circumstances. Or that a combination of some of the approaches may be attractive.

2.2 Distributed databases

A conventional distributed database system (DDBS) [53] can be seen as a special case of the Global Schema approach discussed in section 2.1.1.

An important difference between a multidatabase system (MDBS) and a DDBS is the order in which the integrated schema and the conceptual schemas of the component databases (the local schemas) are created. In a DDBS, the integrated schema is designed first. This is then split up (fragmented) to schemas which will be the conceptual schemas of the component databases. In a MDBS, the local schemas are developed first and independently of each other. They must then undergo a schema integration process and the resulting integrated schema must be defined as a view over the local schemas. In other words, DDBSs are developed top-down whereas MDBSs are developed bottom-up.

DDBSs are homogeneous - the same data model and access language is used for all component databases. In a MDBS, the component database systems may use different data models and access languages. Due to the fact that the component databases are developed independently of each other, MDBSs must also deal with semantic heterogeneity (see footnote 1 on page 3).

A Global Schema MDBS has two kinds of users; some access data directly through a local schema and some access data through the federated schema. In a DDBS, there are no 'local' users. All access goes through the integrated schema.

The component databases in a Global Schema MDBS may be autonomous - they may want to retain complete control over data and query processing at their site. In that case, global integrity constraints and low level coordination of query processing is not possible the way it is in a DDBS.

2.3 Snapshots and materialized views

A central problem in the architectures described in section 2.1 is how to transform and optimize commands against the different types of views into commands against the underlying schemas/views. A query against a federated schema must be split into multiple queries against the underlying com-

ponent schemas. A query against a component schema must be translated to queries in the language of the component database system.

A simple solution to the problem is to use *snapshots*. Data is copied from the underlying database(s) and transformed into the target schema once and for all. Subsequent commands can be processed directly using the copied and transformed data, without any data or command transformations.

This approach is often taken in real settings today. For example, if data from a hierarchical database is to be combined with data from a relational database, the first step is to make a copy of the hierarchical data and transform it to relational structures. These relational structures can then be used together with data from the relational database.

The obvious problem with this approach is that the copied data will become obsolete. Changes to data in the component databases are not reflected in the copies.

Snapshots may still be useful in some situations, especially to improve performance. An interesting idea is to combine snapshots with an active database mechanism [28] [58]. Active rules in the underlying database are defined to trigger when changes occur that affect data that has been copied. The snapshot is then automatically updated to reflect these changes. The usual term for this kind of mechanism is a *materialized view* [27].

2.4 Multi-lingual approaches

In the architectures in section 2.1.1-2.1.3, users access data through different kinds of federated schemas, expressed in the canonical data model. The language used is that of the canonical data model. It is possible to introduce an extra schema layer, on top of the federated schema layer, in which the schemas are expressed in different data models (see figure 2a). Users who are accustomed to a specific data model could then access data through a schema expressed in that specific data model, and could use a language that they are familiar with [13] [31]. They would not have to learn a new language (the language of the CDM).

The drawbacks of this approach are that it increases the complexity of the system (another layer is added), and that some information may be lost. An important quality of the CDM is that it is expressive enough to capture the semantics of all the component databases. If the data model of the user's choice provides less semantic modelling constructs than the CDM, some information will be lost.

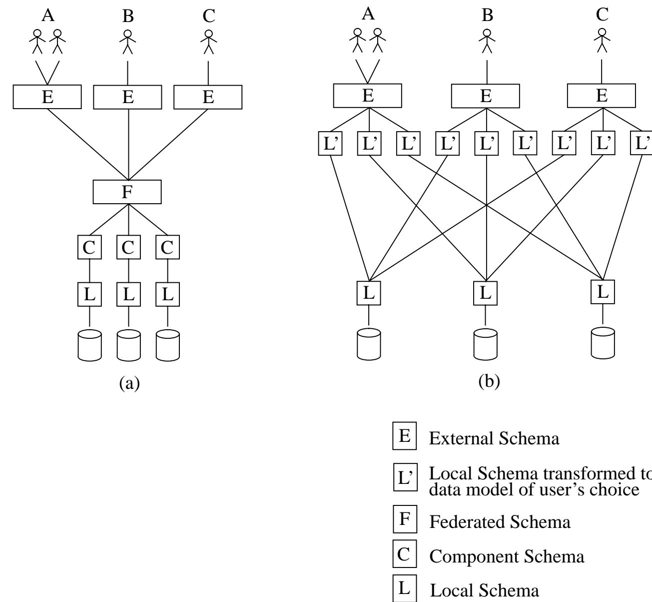


Figure 2: Schema architecture of a Multi-lingual system using the Global Schema approach. Users access data through External Schemas which are described in a data model of the user's choice. Different users prefer different data models. (a) With the use of a CDM. (b) Without the use of a CDM. Compare with figure 1.

2.5 No canonical data model

A central problem in multidatabase systems is how to map data and commands between different data models and languages. The use of a CDM reduces the complexity of this problem. Assume that there are m source data models and n target data models in a multilingual system. If a CDM is used, the number of data-model-to-data-model mappings needed is $m+n$ (the source data models and the target data models must all be mapped to the CDM). Without the use of a CDM, the number is $m*n$ (all source data models must be mapped to all target data models). See figure 2b.

In some sense the absence of a CDM increases the flexibility of a multidatabase system, but it comes at the expense of complexity and duplicated work. The number of data-model-to-data-model mappings increases. Integration solutions (conflict detection and resolution) can not be shared but must be worked out again and again, and in different ways for each data model.

When the number of component systems is very small, and no new components are expected to be added, the use of a CDM may not be needed. A more ad hoc, point-to-point, mapping solution may be justified.

Some work advocates a very flexible type of architecture, in which no integration solutions are worked out in advance [52]. In this sense it is similar to the multidatabase language approach, but here the user does not have to worry about semantic conflicts between component databases. All integration is done by the system (at query time). Semantic knowledge available at query time is used to decide what queries should be sent to what component systems, and how conflicts should be resolved. This includes both data model conflicts (no CDM is used) and semantic conflicts. It is not clear to us how queries are formulated against this type of system. Presumably a very high-level, data model independent language is used (natural language?).

2.6 Combinations

As discussed in section 2.1, there are pros and cons of each of the different architectures. There is no 'best' architecture; different architectures will be best suited for different applications and environments.

By having an architecture that is a combination of some of the 'pure' approaches discussed here, it is sometimes possible to combine the advantages and minimize the disadvantages of the different approaches. For example, by having a multidatabase language as the base for access, but still allowing a multidatabase administration to provide integrated schemas, the flexibility of multidatabase languages is combined with the possibility to share integration solutions. Another example is to have the query and data translation approach as the normal case but complement it with materialized views for applications with high demands on performance.

In a situation where different kinds of multidatabase systems coexist, the next issue is how to make different types of multidatabase systems interoperate. Another problem is how to locate information in very large multidatabase systems [10].

3 Terminology and Standards

Terminology in the area of multidatabase systems is rather confused and inconsistent. Different terms have been used for similar concepts, and identical terms have been used with very different meaning.

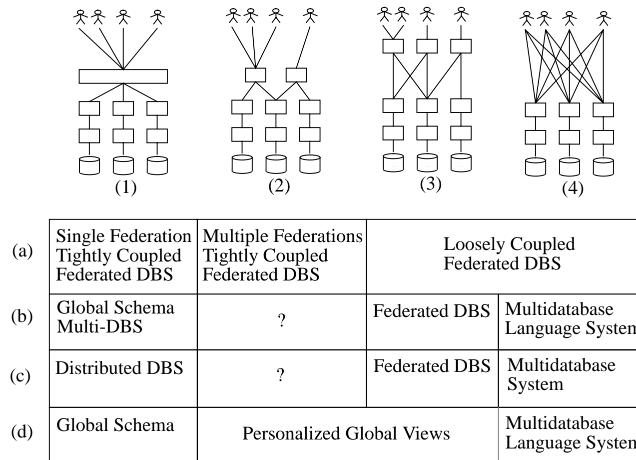
Several overview articles, which all use different terminology, have been written [11] [31] [44] [56]. The most comprehensive of these, and the one most commonly referenced, is the one by Sheth & Larson [56]. We try to follow the terminology from Sheth & Larson and indicate the differences otherwise.

The term *multidatabase system* is starting to become a standard term for the general concept of a system in which it is possible to access data from multiple databases, which may be distributed, heterogeneous, and autonomous. This term is also used here. Other terms that have been used for this general concept include *federated database system*, *heterogeneous database system* and *interoperable database system*. Note that the terms multidatabase system, federated database system, and interoperable database system have sometimes been used for much more restricted classes of systems.

Sheth & Larson use the term *federated database system* for a multidatabase system where the component database systems are autonomous. We do not use the term federated database system in this wide sense, since it was originally used for a much more restricted class of system (and still often is).

Figure 3 gives an overview of terms that have been used in other papers for the four main architecture alternatives described in section 2.1.

Note that in the terminology of [56], a multidatabase system (or federated database system in their terminology) may be called *tightly coupled* and still retain a great deal of autonomy. 'Tightly coupled' in [56] means that a multidatabase administration is responsible for maintaining the integrated schemas. 'Loosely coupled' means that no multidatabase administration is needed, it is the responsibility of the local users (or DBAs) to maintain the integrated schemas.



	(1)	(2)	(3)	(4)
(a)	Single Federation Tightly Coupled Federated DBS	Multiple Federations Tightly Coupled Federated DBS	Loosely Coupled Federated DBS	
(b)	Global Schema Multi-DBS	?	Federated DBS	Multidatabase Language System
(c)	Distributed DBS	?	Federated DBS	Multidatabase System
(d)	Global Schema	Personalized Global Views		Multidatabase Language System

Figure 3: Terminology used by different authors for the four architectures discussed in section 2.1: (1) Global Schema, (2) Multiple Integrated Schemas, (3) Federated, (4) Multidatabase Language. (a) Sheth & Larson [56], (b) Bright et al. [11], (c) Litwin et al. [44], (d) Fankhauser et al. [25]

3.1 Reference schema architecture

Sheth & Larson [56] distinguishes between five levels of schemas in multidatabase systems (see figure 4).

A *local schema* is the conceptual schema of a component database system. Since the component database systems may use different data models, the local schemas may be expressed in different data models.

For each local schema, there is a corresponding *component schema*. The component schema represents the same information as the local schema, but the CDM is used instead of the data model of the component database system. A query against a component schema is translated into queries against the underlying local schema. The results of these queries are then processed to form an answer to the initial query. All component schemas are expressed in the CDM.

For each component schema, one or more *export schemas* may be defined. An export schema represents a subset of the component schema. It defines what part of the component schema is available to a particular group of users.

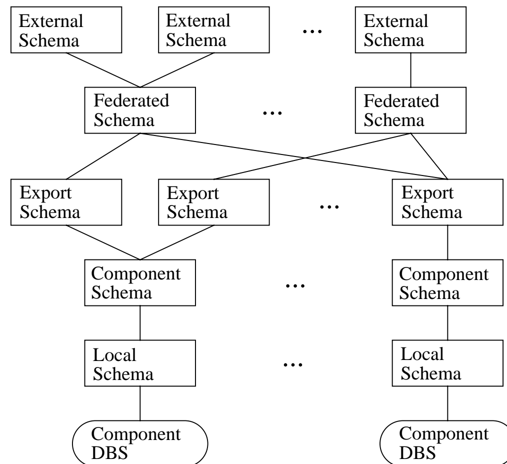


Figure 4: Five-level schema architecture of multidatabase systems (from [56]).

A *federated schema* is an integration of multiple export schemas. It makes it possible to access data from multiple databases as if it was stored in a single database. A query against a federated schema is translated into queries against the underlying export schemas. The results of these queries are then processed to form an answer to the initial query. All federated schemas are expressed in the CDM.

For each federated schema, one or more *external schemas* can be defined. An external schema represents a subset of the federated schema, and the schema may be transformed in some ways to suit the needs of a particular user group. In many ways, it plays the same role as an external schema (view) in the standard three-level ANSI/SPARC schema architecture [61]. The external schema, as described in [56], may be expressed in a different data model than the federated schema.

4 A Look at the Real World

4.1 Legacy systems

In the discussion of multidatabase system architectures above, it was assumed that data that was to be accessed was stored in DBMSs, and that it was described by a schema using some data model. This is the *best* case when an existing information system is to be included in a multidatabase system, and is illustrated in figure 5(a). There is a well-defined interface between applications and data. The use of a DBMS (hopefully) means that data is well-structured and easy to understand. The multidatabase system can be seen as just another application accessing the DBMS.

Unfortunately, this ideal situation is not always the case. Most large organizations use information systems that have evolved during a long period of time. They are typically large, developed using old technology, and very complex. They are often critical to the organization's day-to-day work. The usual term for such information systems are *legacy systems*. Recent attention has been given to the problem of replacing these legacy systems with modern technology, without causing too much disturbances to the organizations day-to-day work [12].

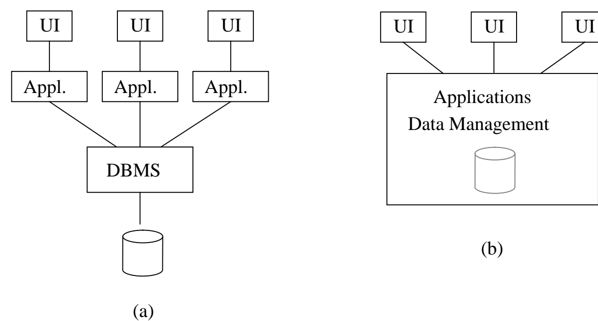


Figure 5: Best case (a) and worst case (b) for including data from an existing information system in a multidatabase system. (a) There is well-defined interface between applications and data. Data is well-structured through the use of a DBMS. (b) There is no clear-cut interface between applications and data. The system has been incrementally extended during a long period of time and has a complex structure. Data is stored in special-purpose structures, often created ad hoc. It is poorly documented.

Figure 5(b) illustrates the *worst* case for including a legacy system in a multidatabase system. There is no modularity in the information system, in particular there is no well-defined interface between applications and data. Due to the long and incremental development history of the system, it is very complex and it is hard to get a complete understanding of all its functionality. It is often poorly documented. No DBMS is used for storage and manipulation of data, instead special-purpose data structures and manipulation procedures are used.

4.2 Commercial state-of-the-art

During the last few years, commercial software that assists in accessing data from heterogeneous data sources has started to emerge. The usual term for such software is *middleware* or *gateways* [42]. Currently available commercial multidatabase systems are much simpler, and much less general than the research systems discussed in previous sections.[7] None of the architectures in section 2.1 applies to current commercial systems, since they are much poorer in terms of integration. Another difference is that the commercial market for multidatabase systems is totally dominated by the relational view of data. The aim of most middleware products is to let applications use different relational databases together [65] [66].

The most advanced middleware products provide an *API* (Application Programming Interface) which gives programmers a uniform SQL interface to different relational (and some non-relational) databases. Programmers do not have to worry about different SQL dialects, different APIs, and different network protocols. However, the products give very little help with integration problems. There are no federated schemas, and no real multidatabase language. Typically, queries are not allowed to span multiple databases (no joins between tables from different databases are allowed). All integration has to be performed in the application program. This is usually what it means when a supplier of middleware claim that they provide 'seamless access to heterogeneous data sources'.

Another large group of related software are data conversion tools [65]. They perform a translation (once and for all) of data from one format to another. Data can be converted between spreadsheets, different types of databases, ASCII files, etc.

Some important acronyms are [63]:

SAG (SQL Access Group). This is a consortium of more than 40 leaders in the database industry. They have among other things developed a standard for SQL APIs.

ODBC (Open Database Connectivity). This is an API which is a superset of

7. On the other hand, the commercial systems really exist, something which is not always the case with research systems.

the SAG API. It has been developed by Microsoft. The ODBC can be used for all relational DBMSs (for which an ODBC driver exists). A lot of products use ODBC, also from other vendors than Microsoft.

IDAPI (Integrated Database Application Programming Interface). This is a competing standard for SQL APIs, which is to be released by Borland. It contains different extensions to the SAG API.

5 Canonical Data Model

Different data models are differently suitable as the CDM in multidatabase systems.[8] During the 80's, the relational data model was very dominantly used as the CDM, although some projects used the E-R model or a functional data model. As will be shown in this section, the relational data model is not very suitable as CDM. Recent research often use an object-oriented data model as the CDM, which is much more promising.

The most important property of a CDM is its semantic expressiveness, i.e. what constructs it provides for modelling data. There are two main reasons for this:

- It must be possible to capture all of the semantics of the data sources in the CDM. Ideally, the expressiveness of the CDM should be greater than, or equal to, the expressiveness of all the data models of the component database systems. Otherwise, some information will be lost when data is transformed to the CDM.
- Semantically rich schemas makes interdatabase correspondences easier to find. If the CDM is semantically richer than the data model of a component database system, the transformation of data to the CDM includes a semantic enrichment process [14]. The more expressive the CDM is, the more semantics it is possible to capture in a component schema. This makes it easier to understand the relationship between different component schemas, which simplifies subsequent integration.

Different so called *semantic data models*, data models with great semantic expressiveness, have been proposed since the 1970's. They have mainly been used as a database design tool, not as the conceptual data model of DBMSs. With object-oriented DBMSs, this is starting to change.

Section 5.1 gives an overview of semantic modelling constructs. Section 5.2 discusses other aspects which concern the suitability of data models as the CDM.

---

8. Sometimes the term 'data model' is used for a schema modelling a specific domain, e.g. 'the marketing data model'. This is not what is meant by a data model in this thesis. Data model, as the term is used here, means 'a set of concepts that can be used to describe the structure of a database' [23]. Examples of data models are 'the relational data model' and 'the E-R data model'.
Another question is whether the operations operating on data are a part of the data model, e.g. whether SQL or the relational algebra can be considered part of the relational data model. Usually, the operations *are* considered part of the data model, but not a *necessary* part. For example, the E-R data model is useful without operations during database design. This is also the view taken here.

5.1 Semantic modelling constructs

Providing constructs for conceptual modelling of a domain is central to both artificial intelligence, databases, and programming languages. Some semantic modelling constructs keep recurring in all these areas. We have classified them in two categories; essential and complementary. Essential constructs are the most important. They are a part of practically all high-level (semantic) data models [5] [16] [32] [55]. The complementary constructs make it possible to capture even more of the semantics of a domain in the database, but they are not used as extensively as the essential constructs.

5.1.1 Essential

Types and instances. Objects which share structure and behaviour (for example, all documents have a title and an author, and they can be printed) can be grouped together. The common structure and behaviour is defined by a *type*. Objects are *instances* of types (for example. the thesis you reading right now is an instance of the type document). Sometimes the terms *classes* and *individuals* are used instead of the terms types and instances.[9]

A related concept is that of *object identity.* Each object has a unique, immutable identifier which can always be used to refer to it. This means that objects have an existence which is independent of the values of their attributes, something which is different from value-based data models such as the relational data model.[10]

Generalization/specialization. Types can be organized as *subtypes/supertypes* to each other. For example, the type conference paper is a subtype of the type document.[11] The subtype *inherits* all the properties of the supertype, and may have additional properties that the supertype does not have. For example, conference papers have all the properties that documents have, but they also have the property 'published in', which gives the conference proceedings that the paper was published in. A particular conference paper is an instance of the type conference paper, and it is also (because of the subtype/supertype relationships) an instance of the type

9. Sometimes the terms 'type' and 'class' are used as synonyms, sometimes they are distinguished from each other (with subtle differences). Often, 'type' refers to the *intension* of a group of objects (much in the same way as an abstract data type in programming languages), whereas 'class' refers to the *extension*.

10. If two tuples in the relational data model have the same value for all attributes, the tuples are considered identical.

11. And document is a supertype of conference paper. Another way to put it is that conference paper is a specialization of document, and that document is a generalization of conference paper.

The subtype/supertype relationship between types is often called the *is-a* relationship (a conference paper 'is a' document).

document (and all of its supertypes).[12]

Complex objects. Complex objects can be built by applying constructors to other objects. The two most important ways of building complex objects are through *aggregation* and *grouping*.

Aggregation (also called 'cartesian aggregation') means that a new type is created as the cartesian product of other types. For example, the type 'address' is an aggregation of the types 'city', 'street', and 'zip code'. Complex objects of this kind are created by applying the *tuple* or *record* constructor on existing objects.

Grouping (also called 'collection', 'association', or 'cover aggregation') means that a set of objects of some existing type are grouped together. For example, the 'drives' attribute of a person is a set of objects of the type 'car'. Complex objects of this kind are created by applying the *set* or *bag* constructor on existing objects.

5.1.2 Complementary

Different kinds of generalization/specialization. It is possible to distinguish between different kinds of generalizations/specializations [23]. A generalization/specialization can be *disjoint* or it can be *overlapping*. It is disjoint if all the subtypes are disjoint (an object can be an instance of only one of the subtypes). Otherwise, it is overlapping. For example, consider a type employee which has two subtypes - secretary and salesman. The generalization/specialization is disjoint if it is impossible for an employee to be both a secretary and a salesman. If it *is* possible to be a secretary and a salesman at the same time, then the generalization/specialization is overlapping.

A generalization/specialization can also be *total* or *partial* (this is orthogonal to the disjoint/overlapping criterion). It is total if every instance of the supertype must also be an instance of some of the subtypes. Otherwise, it is partial. For example, if it is impossible to be 'just an employee' (there are no direct instances of the type 'employee'), one must be either a secretary or a technician, then the generalization/specialization is total. If it *is* possible to have direct instances of the type employee, then the generalization/specialization is partial.

Multiple inheritance. Multiple inheritance is supported if a type can have more than one supertype. For example, the type teaching assistant is a subtype of both student and teacher.

Other types of complex objects. By applying other constructors, different types of complex objects than the ones described above can be created. An

12. These different kinds of instance-of relationships can be distinguished by saying that a conference paper is a *direct instance* of the type conference paper, whereas it is an *instance by generalization* of the supertypes of conference paper.

example of this is *grouping with order*. By applying the *list* or *array* constructor on existing objects, it is possible to capture the concept of order between objects.

Part-of semantics. Sometimes when a complex object references another object, the relationship between the objects may be characterized as a *part-of* relationship (the referenced object is 'part of' the complex object) [37]. An object that is built up from other objects through part-of relationships is sometimes called a *composite object*. Part-of references have different semantics than 'general' references. For example, two of the attributes of a car object are 'engine' (an instance of the type engine) and 'owner' (an instance of the type person). The 'engine' attribute is a part-of reference. The 'owner' attribute is a general reference. If the car object is deleted, the engine object should also be deleted, whereas the person object should not.

5.2 Other aspects

The suitability of different data models as CDM is discussed by Saltor et al. in [55]. Besides semantic expressiveness, they mention *semantic relativism* as an important property of the CDM. By semantic relativism of a data model they mean 'the power of its operations to derive external schemas'.[13] The relational data model is an example of a data model with high semantic relativism, since a powerful view definition capability is provided (e.g. in SQL).

If the five-level schema architecture of [56] is used (see section 3), the view definition capabilities of the CDM must be used for the export schemas and the external schemas. To define a federated schema, it must also be possible to define a view over multiple export schemas. Note that using a declarative view definition language (like SQL) is only one of the possible approaches for this. In [50], a 'step-by-step' editing process is used to transform the export schemas into a federated schema. The mapping between the federated schema and the export schemas is derived from the editing process. In [59], a semi-automatic process is used, in which the DBA declares what correspondences exist between objects in different export schemas. These correspondences are then given as input to a tool which automatically creates the federated schema and the mappings between it and the underlying export schemas.[14]

In the context of semantic relativism, Saltor et al. also mention the advantage of having a single basic modelling construct. The relational data model is good in this respect, since the relation is the only modelling construct. The E-R model is bad in this respect, since there are two basic constructs; entities and relationships. This causes problems during schema integration,

13. The term 'external schema' is used in the ANSI/SPARC architecture [61] sense.
14. The DBA assists the tool when it faces unresolvable conflicts.

since an object may be modelled as an entity in one schema and as a relationship in another schema. We believe that having a single basic modelling construct is not so important. It may simplify schema integration to a small extent, but the problems of semantic heterogeneity will always exist. A single fact may always be modelled in many different ways, even if there is a single basic modelling construct. For example, in [35] Kent describes 36 ways to model the simple fact that salesmen serve territories in a relational-like data model.

6 The AMOS Multidatabase System

As discussed in section 2.1, there are advantages and disadvantages of all the 'pure' architecture alternatives. The AMOS architecture is an attempt to combine the advantages of the different approaches while minimizing the disadvantages.[15] Using the terminology from section 2.1, the AMOS architecture is a combination of the multiple integrated schemas approach, the federated approach, and the multidatabase language approach.[16]

The schema architecture and software components of AMOS are described in section 6.1. The AMOS data model, which is used as the CDM, is a functional and object-oriented data model. Section 6.2 presents the structural part of the AMOS data model. Operations on data (the AMOSQL language) is described in section 6.3. Section 6.4 discusses the suitability of the AMOS data model as a CDM by comparing it to the criterions given in section 5.

6.1 Architecture

Figure 6 shows the AMOS schema architecture. The project concentrates on the two most important types of mappings - the ones providing the component schemas (mappings from different data models to the CDM) and the ones providing the federated schemas (mappings from multiple schemas in the CDM to an integrated schema in the CDM). Therefore, the export and external schema levels from Sheth & Larson (see section 3.1) are not included in the AMOS architecture.

The local schemas are expressed using the data model of the component database systems. A component schema represents the same information as the underlying local schema, but the CDM is used instead of the data model of the component database system. A federated schema is an integration of multiple component schemas.

---

15. AMOS (Active Mediators Object System) [24] is an umbrella project for database research at the University of Linköping. The central parts of the project has so far been active databases and multidatabase systems. In the descriptions of the AMOS architecture in this thesis, only the components which have something to do with multidatabase issues are represented.

16. Using the terminology of Sheth & Larson [56], the AMOS architecture is a combination of a loosely coupled federated database system and a tightly coupled federated database system (with multiple federations).

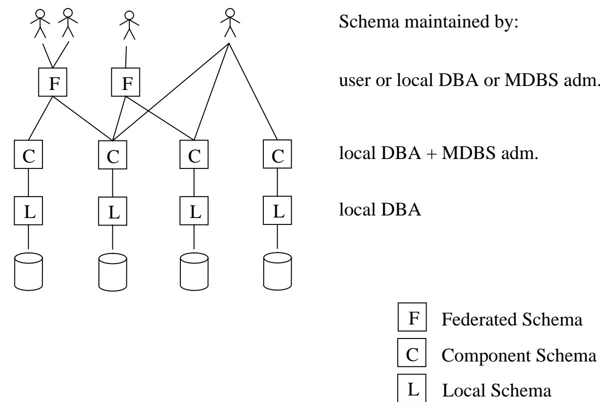


Figure 6: The AMOS multidatabase system schema architecture.

The basic way to access data in AMOS is through a multidatabase language. The multidatabase language provides a very flexible way to work with data from multiple data sources. Any combination of data can be used together at any time without the need to define an integrated view of data (a federated schema) in advance.

Federated schemas can be defined to enable sharing and reuse of integration solutions. The federated schemas can be developed and maintained by users or by the DBAs at the site of the users. Or they can be maintained by a multidatabase administration to enable a wider sharing of integration solutions.

The multidatabase language will have view definition capabilities. This means that it can be used to define the federated schemas. The relationship between the federated schema and the underlying component schemas is defined with declarative view definition statements.

Recall that the CDM used in AMOS is an object-oriented data model. A completely general view mechanism should be able to map *from* all kinds of constructs in the data model *to* all kinds of constructs in the data model. Most proposals for object views are somewhat limited in this respect. This is further discussed in section 6.3.

Figure 7 shows the software components of the AMOS architecture.

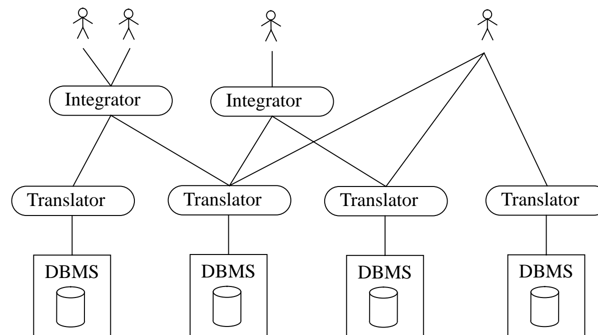


Figure 7: Software components in the AMOS multidatabase system architecture.

Translators implement the mappings between local schemas (expressed in the data models of the component database systems) and the corresponding component schemas (expressed in the CDM). There is one kind of Translator for each *kind* of data source. A query sent to a Translator is transformed to calls to the underlying data source. The results of these calls are then processed to form an answer to the initial query. A Translator can be used by one or more integrators or directly by users or application programs.

Integrators implement the mappings between component schemas and federated schemas. A query sent to an Integrator is transformed into several queries against the underlying Translators.[17] The results of these queries are then processed to form an answer to the initial query.

To access data, it is not necessary to use an Integrator. Queries can be put directly against Translators using the multidatabase language. The multidatabase language is also used to define the mapping between Integrators and Translators (the mapping between federated and component schemas).

All Translators and Integrators have a local AMOS database with full database management capabilities. Part of the schema that a Translator/Integrator presents to its users is stored directly in the local database and part of it is a view of data which resides in foreign databases. This will be discussed further in section 13.

Related work of particular interest are the Multibase [40] and Pegasus [2] [3] projects.

Multibase has an architecture which is similar to the AMOS architecture,

17. A federated schema may be defined over other federated schemas (not only over component schemas). Hence, a query sent to an Integrator may actually result in queries being sent to other Integrators (not only to Translators).

although the role of the multidatabase language is somewhat different. The language which is used for defining the federated schemas can be seen as a multidatabase language. However, this language can not be used directly by users or applications to access multiple component schemas. Only the DBA uses it when the federated schemas are defined. Multibase uses the functional data model in [57] as the CDM and DAPLEX as the data manipulation language. The AMOS data model is a derivative of this data model, but an important difference is that the AMOS data model is object-oriented; queries can return OIDs. Another difference is the role of the Translator. In Multibase, the mapping between the local schema and the component schema is the simplest possible. All semantic enrichment is performed in the Integrator.

The Pegasus project uses the IRIS data model [26] [48] as their CDM and an extension to OSQL as the data manipulation language. The main difference to AMOS is architectural. A Pegasus server performs both translation and integration, whereas in AMOS this is separated in two kinds of modules. Each AMOS Translator only needs to know the data model of one data source and how to map this to the CDM. The Pegasus server must understand all underlying data models and must have language constructs for mapping each of these data sources to the CDM.

6.2 The AMOS data model

The AMOS data model is a functional and object-oriented data model which is based on the IRIS data model [26] [48], which in its turn is based on the functional data model in [57] ('DAPLEX').[18]

There are three basic constructs in the AMOS data model; *objects*, *types* and *functions*. Objects are used to model entities (concrete or abstract) in the domain of interest. Types are used to classify objects; an object is an instance of one or more types. Properties of objects and relationships between objects are modelled by functions. For example, Sweden and Norway are modelled by objects which are instances of the type *country*. The number of people living in each country is modelled by a function *inhabitants* which takes a *country* object as argument and returns an integer.[19]

The set of objects that are instances of a type is called the *extension* of the type. Types are divided into *literal* types and *surrogate* types.[20]

---

18. The AMOS data model is *very* similar to the IRIS data model as it is described in [48].
19. Types and functions are first class objects. All types are objects that are instances of the type *type*. All functions are objects that are instances of the type *function*.
20. Some authors make the distinction between *immutable* and *mutable* objects. They correspond to instances of literal and surrogate types, respectively.

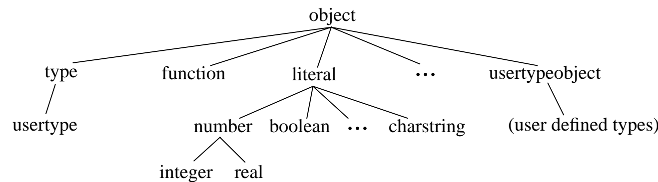


Figure 8: Part of the AMOS subtype/supertype graph. Each line represents a subtype/supertype relationship. *object* is the most general type.

The extension of a literal type is fixed (often not enumerable), and instances of a literal type are self-identifying; no extra object identifier is needed. Examples of literal types are integers, character strings, and floating-point numbers.

Surrogate types and instances of surrogate types are created by the system or by users. Instances of surrogate types are identified by a unique, immutable, system-generated *object identifier (OID)*.[21] Examples of surrogate types are person, document, country etc.

Types are organized in a subtype/supertype graph. Figure 8 shows part of the type graph of AMOS. The most general type is *object*; all other types are subtypes of *object*. User defined types are subtypes of a special type called *usertypeobject*.

All types are instances of the type *type*. User defined types are also instances of the type *usertype*.

Figure 9 shows an example of subtype/supertype relationships between user defined types. A type inherits all the properties (i.e. functions) of its supertypes. For example, *conference_paper* inherits all the properties of *document*, which in its turn inherits all the properties of *usertypeobject* etc. A type can be a direct subtype of more than one supertype. For example, *teaching_assistant* is a direct subtype of both *student* and *teacher*, and therefore inherits the properties of both these types (*multiple inheritance*).[22]

---

21. OIDs can be *logical* or *physical* [15]. Logical OIDs do not contain the actual address of the object. The address of an OID is retrieved using an index. OIDs in AMOS are logical.
AMOS uses a monotonically increasing counter to produce unique OIDs.
22. We do not discuss the problems of multiple inheritance (e.g., what *salary* function should *teaching_assistant* inherit if it is defined for both *student* and *teacher*?) in this thesis.

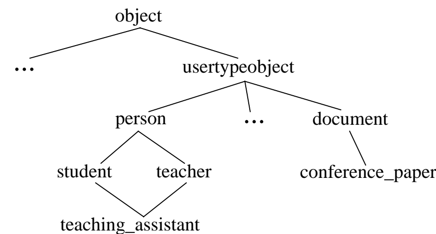


Figure 9: Example of user defined types in the AMOS subtype/supertype graph.

An object is a *direct instance* of one or more types. For example, an object may be an instance of the types *vegetarian* and *employee* at the same time.[23] The object is also (because of the subtype/supertype relationships) an instance of all supertypes of these types (see footnote 12 on page 25).

Properties of objects and relationships between objects are modelled by functions. For example, to model the fact that suppliers supply parts to departments, a function *supply* may be used. The function takes two arguments; a department object *d* and a supplier object *s*. It returns the part objects that *s* supplies to *d*.

The relationship between arguments and results for a function is called the *extension* of the function. A function is implemented in one of three different ways; it may be *stored*, *derived*, or *foreign*. For stored functions, the extension is stored directly in the database. A derived function uses the AMOSQL query language to calculate the extension. A foreign function is implemented in a general programming language, such as C or LISP.

The *signature* of a function is the name of the function together with its argument types and result types. A function can have zero, one, or more argument types and zero, one, or more result types.

A function that can have multiple results for the same argument(s) is called a *multivalued* function. For example, the function *hobby(employee)->charstring* is a multivalued function since an employee can have more than one hobby.

23. In most object-oriented data models, an object can only be a direct instance of one type. The employee/vegetarian example would be handled by introducing a new type *vegetarian_employee*, which would be a subtype of both *vegetarian* and *employee*. Vegetarian employees would then be direct instances of this new type. The main problem with this approach is that it can lead to a complex type graph with a large number of types that are not very natural.

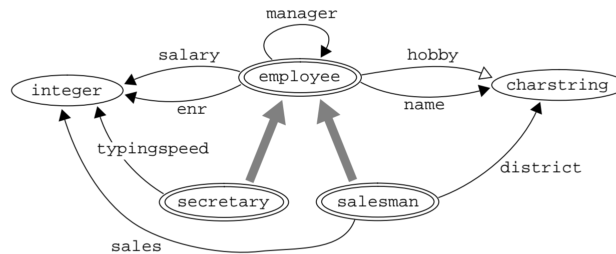


Figure 10: Graphical notation for AMOS schemas. Types are represented by ovals; single ovals denote literal types, double ovals denote surrogate types. A function is represented by a thin arrow from its argument type(s) to its result type(s). A hollow arrow-head denotes a multivalued function. Bold arrows represent subtype/supertype relationships (is-a relationships).

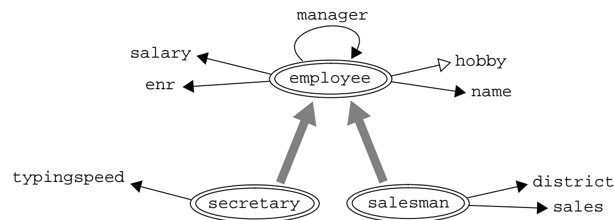


Figure 11: Simplified graphical notation for AMOS schemas (equivalent with the schema in figure 10). Literal types are not represented.

Functions can be *overloaded* - the same name can be given to functions defined on different types. When an overloaded function name is used, the right function is chosen by looking at the types of its arguments and results. This is called *function name resolution* and the chosen function is called the *resolvent*. Functions are not associated to *one* type (as methods usually are in object-oriented programming languages). This means that functions can be overloaded not only on their first argument type, but on *all* argument and result types.[24]

Figures 10 and 11 illustrate the graphical notation for AMOS schemas that is used in this paper. The schema in these figures will be used as an example throughout the paper.

24. In the current implementation of AMOS, however, overloading *is* only allowed on the first argument type.

6.3 The AMOS query language

The data definition and manipulation language of AMOS is called AMOSQL (AMOS Query Language). AMOSQL is based on OSQL, the language used in the IRIS DBMS [26] [48].

Functions can be used directly to answer simple queries. Example:[25]

```
amos 17>plus(4,7);
<11>
amos 18>sqrt(9);
<3>
<-3>
```

For more general queries, a construction with a syntax similar to that of SQL can be used. The syntax is:

```
select <result>
for each <type declarations for local variables>
where <condition>
```

Example:

```
amos 19>select i+5 for each integer i
        where i=sqrt(9) and i>0;
<8>
```

Functions can be used in the forward direction or in the backward direction. Here the function *hobby* is used in the forward direction:[26]

```
hobby(:e1);
```

Here it is used in the backward direction:

```
select e for each employee e where hobby(e)='sailing'
```

A function returns a *bag of tuples*. The semantics of nested function calls is as follows (this is called 'DAPLEX semantics' here, since it is based on the semantics of the DAPLEX language [57]). When a function is called with a bag as argument, the function is applied to all the members of the bag (one at a time). The result of the function call is the union of all the results of applying the function to the different bag members. An example:

Consider a function *parent(person)->person* that takes a *person* object as input and returns the *person* objects representing the parents of that person. An example extension of the function is shown in table 2.

---

25. 'amos nr>' is the prompt of the AMOS interpreter. The result of a function call is a bag of tuples (this is discussed later in this section). The AMOS interpreter displays the result tuples with one tuple per row.
Arithmetic functions can also be used with infix notation. For example:

```
amos 17>4+7;
<11>
```

26. *:e1* is an environment variable that is bound to an *employee* object. We use the convention that names of environment variables begin with a colon (:).

p1	p2
:adam	:bertil
:adam	:cecilia
:bertil	:david
:bertil	:edla
:cecilia	:filip
:cecilia	:greta

Table 2: Example. The extension of a function *parent(person p1)->person p2*.

Now consider the following nested function call:[27]

```
parent(parent(:adam));
```

The result of applying *parent* to *:adam* is a bag of two tuples (each consisting of a single object):

```
[<:bertil>, <:cecilia>]
```

When *parent* is called with this bag as argument, it is applied first to *:bertil* which gives the bag [<:david, :edla>] as result, and then to *:cecilia* which gives the bag [<:filip>, <:greta>] as result. The final result of the nested function call therefore is:

```
[<:david>, <:edla>, <:filip>, <:greta>]
```

(End of example.)

There is one exception to the DAPLEX semantics, namely aggregation operators. For example, the *sum* function, which gives the sum of all members of a bag, is applied once on all the members of the bag, not one time for each member (which would not make much sense). A function is given this semantics if the type of its only argument is declared as 'bag of ...'. For example:

```
create function sum(bag of integer)->integer as ...
```

New types are created with the 'create type ...' statement. Example:

```
create type person;
create type employee subtype of person;
```

27. The input to functions are also bags of tuples. Function calls like 'parent(:adam)' can be seen as a shorthand for 'parent([<:adam>])'. Actually, this 'shorthand' is the only way to express function calls in the current implementation... For example, function calls like 'plus([<4,7>,<2,3>])' (which should return the bag [<11>,<5>]) are not possible.

Instances of a type are created with the ’create <*type*> instances ...’ statement. Example:

```
create employee instances :adam, :bertil, :cesar;
```

The type membership of instances can be changed with the ’add type ...’ and ’remove type ...’ statements. Example:

```
add type vegetarian to :adam;
```

New functions are created with the ’create function ...’ statement. Example:

```
create function foo(integer a)->charstring b as stored;
create function fie(real a, integer b)->real c as stored;
create function fum(charstring a)-><real b, integer c>;
```

Derived functions are defined in terms of other functions. Example:

```
create function emp_w_hobby(charstring h)->employee e as
  select e for each employee e where hobby(e)=h;
```

Foreign functions are implemented in a general programming language, such as C or LISP. This is not discussed further here.

The extension of a stored function is defined with the ’set ...’ command. Example:

```
set salary(:adam)=10000;
set hobby(:adam)='boxing';
```

The extension of multivalued stored functions can be changed with the ’add ...’ and ’remove ...’ commands. Example:

```
add hobby(:adam)='sewing';
remove hobby(:adam)='sewing';
```

During function name resolution, AMOS uses variable declarations to choose the resolvent function (*early binding*). For example, suppose that the functions *manager(employee)->employee* and *manager(department)->employee* are defined. Consider the AMOSQL query:

```
select manager(d) for each department d;
```

The resolvent function *manager(department)->employee* is chosen since the variable *d* is of the type *department*.

Sometimes, function name resolution can not be performed at compile-time but must be delayed until run-time (*late binding*). For example, suppose that salary is an overloaded function and that two resolvents exist - *salary(employee)->integer* and *salary(salesman)->integer*. The salary of a salesman is defined as his/her salary as an employee plus a bonus that depends on his/her sales. Now consider the definition of the following derived function:

```
create function all_salaries()->integer as
  select salary(e) for each employee e;
```

The function salary should be applied to all employee objects. It is not pos-

sible to replace the salary function with any of its resolvents at compile-time, since different resolvents should be chosen for different employees. If the employee is a salesman, the function *salary(salesman)->integer* should be chosen, otherwise the function *salary(employee)->integer* should be chosen.

AMOS automatically checks the subtype/supertype graph to decide whether early binding is possible or not. If a resolvent function *can* be chosen a compile-time, it *will* be. In the case of conflicts, the resolution of a function name is delayed until run-time. Sometimes addition or removal of functions will cause (automatic) recompilation of other functions. For example, suppose that the function *salary(salesman)->integer* is deleted. Now, early binding would be possible in the all_salaries function and it will therefore be recompiled.

The only view mechanism that the current version of AMOSQL provides is the concept of derived functions. This is equally powerful as the views provided in relational databases, but object views should include something more. A completely general view mechanism should be able to map *from* all kinds of constructs in the data model *to* all kinds of constructs in the data model. For example, in the AMOS data model it should be possible to have types in the view which correspond to objects or functions in the base schema, or vice versa. Most proposals for object views [1] [9] [29] are limited in this respect. We are only aware of one paper [18] which discusses these issues. Note that the relational view mechanism is not completely general either [39] [45]. For example, it is not possible to have relations in the view which correspond to values in the base schema.

The current version of AMOSQL has no multidatabase language capabilities.

6.4 The AMOS data model as a CDM

The AMOS data model provides all the semantic modelling constructs characterized as essential in section 5, and some of the complementary.

Types and instances. Objects are classified by types. Objects have a unique object identity and are instances of one or more types.

Generalization/specialization. Types are organized as subtypes/supertypes and subtypes inherit all functions that are defined on the supertypes.

Complex objects. Aggregation is supported through the notions of types and functions.

For example, the aggregation *address* discussed in section 5 is created by defining a new type *address*, and the functions *city(address)->city*, *street(address)->street*, and *zip_code(address)->zip_code*. Explicit creation of tuple objects is also possible. Grouping is supported through the notion of multivalued functions. For example, the *drives* grouping dis-

cussed in section 5 is created by defining a multivalued function *drives(person)->car*. Explicit creation of set and bag objects is also possible.

Different kinds of generalization/specialization. This is not supported by the AMOS data model.

Multiple inheritance. An AMOS type can have more than one direct supertype. A related feature of the AMOS data model is that an object can be a direct instance of more than one type.

Other kinds of complex objects. Order between objects can be captured with the *vector* data type.

Special part-of semantics. This is not supported by the AMOS data model.

Semantic relativism. A view mechanism with an expressiveness equivalent to that of relational views is supported through the concept of derived functions. We are working on a more general view mechanism for the AMOS data model. See also section 6.3.

7 Summary of Part I

A multidatabase system is a system in which it is possible to access and update data residing in multiple databases. The databases may be distributed, heterogeneous, and autonomous.

We gave an overview of different kinds of multidatabase system architectures and discussed their relative merits. We also discussed standards in the field and contrasted the terminology used by different authors with each other.

A central problem in multidatabase systems is that of data model heterogeneity; the fact that the participating databases use different conceptual data models. It is common to use a canonical data model (CDM) to handle this. When a CDM is used, the schemas of the participating databases are mapped to equivalent schemas in the CDM. We discussed what properties a data model should have to be suitable as a CDM. Object-oriented data models are attractive candidates.

We then presented the AMOS multidatabase system architecture, which is designed with the purpose of combining the advantages and minimizing the disadvantages of the different kinds of proposed architectures. The AMOS data model, which is used as the CDM, is a functional and object-oriented data model.

Part II Object Views of Relational Data

8 Introduction

The topic of this second part of the thesis is *object views of relational data*. Such a view mechanism makes it possible for users (end-users or application programs) to transparently work with data in a relational database as if it was stored in an object-oriented database. Queries against the object view are translated into queries against the relational database. The results of these queries are then processed to form an answer to the initial query. Update commands to the object view result in updates to the relational database.[28] All this is transparent to users of the view. In this thesis, we concentrate on *access* to relational databases via object views, not updates.

The context in which object views of relational data is discussed in this thesis is that of multidatabase systems. As was discussed in part I, most multidatabase systems use a CDM to deal with the problem of data model heterogeneity. It is generally agreed that object-oriented data models are appropriate as the CDM in a multidatabase system (see section 5). If an object-oriented CDM is used, the different local schemas must be mapped to object-oriented structures in the component schemas, i.e. object views must be established for the different types of component databases. Since relational databases have such a dominating position on the database market, techniques for developing object views of relational databases are especially important.[29]

Note that the discussion in this part of the thesis is not dependent on the AMOS architecture. This kind of object view is needed in any multidatabase system that uses an object-oriented CDM.

The use of object views of relational data is not limited to multidatabase systems. A semantically richer view of data makes it easier for users to understand the meaning of data. It also decreases the impedance mismatch between the database and object-oriented programming languages.

This kind of view will also be useful during *legacy system migration* [12]. Suppose that a relational database is to be replaced with an object-oriented database. To ensure a graceful transition, it will be helpful to start by letting the new application work with an object view of the relational database. The environment can then be incrementally changed so that more and

28. Only a certain class of updates to the view can be unambiguously translated to updates to the relational database. This is a general problem of views [23].

29. Relational databases totally dominate the *current* database market, but a large part of the database systems that are running today are old and belong to the first generation of database systems (hierarchical and network). It has been estimated that approximately 30 % of the database systems running today are relational.

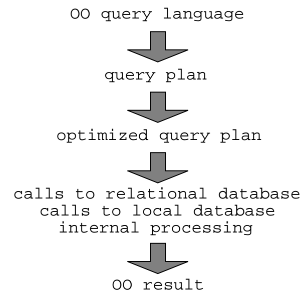


Figure 12: Processing queries against an object view of a relational database.

more data is stored physically in the object-oriented database and is not accessed through the view.

In section 6 we mentioned the Pegasus project [2] [3] which has a lot in common with the multidatabase part of the AMOS project. In the area of object views of relational data, the Pegasus project has concentrated on techniques for automatic generation of the *schema* of the object view [4]. Work of this kind has also been reported in [34] [49] [51] [64]. The focus of our work is on *query processing*. We show how queries against the object view are translated, optimized, and executed. The different query processing steps are illustrated in figure 12.

We will end this introductory chapter by introducing an example (section 8.1) which will run through the rest of the thesis, and with a short introduction to query processing in object views of relational data (section 8.2). The rest of this part of the thesis (II) is organized as follows:

Sections 9 and 10 give short overviews of the relational data model and the central concepts of object-oriented data models. Section 11 concerns the relationship between the relational data model and object-oriented data models. It discusses how schemas in an object-oriented data model can be mapped to schemas in the relational data model, and vice versa. It also introduces a normal form for representing subtype/supertype relationships in relational schemas.

To be able to discuss query processing in object views of relational data, we first give an overview of query processing in object-oriented databases (section 12). We are then ready for the central part of this thesis; section 13. This section discusses techniques that are needed to provide object views of relational data, and shows how this is implemented in AMOS. Section 14, finally, gives a summary of this part of the thesis.

employee

enr	name	salary	manager
314	anna	20000	265
159	bertil	15000	314
265	cesar	25000	NULL
358	doris	13500	314

emp_hobbies

employee	hobby
314	sailing
314	golf
159	golf
265	tennis
265	fishing
265	golf
358	tennis

secretary

enr	typingspeed
358	1100

salesman

enr	district	sales
159	kisa	70
314	rimforsa	40

Figure 13: The company database (example of a relational database).

8.1 The company example

Throughout this part of the thesis we will use the same example of a relational database. We will use this example to show the relationship between a relational database and the corresponding object view, and to show how queries against the object view are processed.

The example relational database is shown in figure 13. It stores information about the employees of a company.[30] The employee number (enr), name, salary, and manager for each employee are stored in the *employee* table. Their hobbies are stored in the *emp_hobbies* table. An employee can have more than one hobby. The *secretary* table contains the typingspeed for each secretary. The *salesman* table stores which districts the salesmen work in, together with how much they have sold (sales). A salesman only works in one district.

Figure 14 shows the schema of the corresponding object view of the company database.[31] There are three types in the view; employee, secretary, and salesman. secretary and salesman are subtypes of employee. The properties of employees, secretaries, and salesmen are modelled by functions.

30. The examples were designed to be as simple as possible, while still illustrating the basic features of relational and object-oriented databases, and the principles of processing queries against them. This has come at the expense of the examples not being very realistic.

31. We will use the AMOS data model as our example of an object-oriented data model. The AMOS data model and the notations used to illustrate AMOS schemas and extensions were introduced in section 6.2.

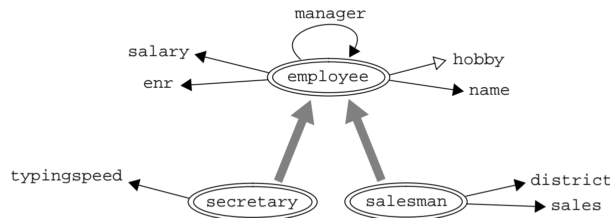


Figure 14: Schema of the object view of the company database.

Figures 15 and 16 show the extension of the object view (textual and graphical representations, respectively). Note that the extension of the view is not physically stored anywhere. It is computed every time it is used. However, to a user of the view, it behaves exactly as if the extension had been physically stored.

If you examine the extension of the relational database, you will find that it stores information about four employees. Two of these ('anna' and 'bertil') are salesmen and one ('doris') is a secretary. One of them ('cesar') is neither a salesman nor a secretary, but still an employee. Accordingly, there should be four objects in the object view (:e1, :e2, :e3, and :e4). Two of them should be direct instances of salesman, one a direct instance of secretary, and one a direct instance of employee.

Informally, the semantics of the mapping between tuples in the relational database and objects in the object view is as follows. There is one object for each tuple in the employee table. The primary key enr is used to define the correspondence between tuples and objects. For example, the enr 314 corresponds to the object :e1. All objects are instances of the employee type. An object is also an instance of the type secretary (salesman) if there is a tuple in the secretary (salesman) table with the enr that corresponds to the object. For example, the object :e1 is an instance of the type salesman, since there is a tuple with enr=314 in the salesman table.

```
direct_instance_of(:typeEmployee)=:e3
direct_instance_of(:typeSecretary)=:e4
direct_instance_of(:typeSalesman)=:e2
direct_instance_of(:typeSalesman)=:e1

enr(:e1)=314                   enr(:e2)=159
name(:e1)='anna'               enr(:e2)='bertil'
salary(:e1)=20000              salary(:e2)=15000
manager(:e1)=:e3               manager(:e2)=:e4
hobby(:e1)='sailing'           hobby(:e2)='golf'
hobby(:e1)='golf'              district(:e2)='kisa'
district(:e1)='rimforsa'       sales(:e2)=70
sales(:e1)=40

enr(:e3)=265                   enr(:e4)=358
name(:e3)='cesar'              name(:e4)='doris'
salary(:e3)=25000              salary(:e4)=13500
hobby(:e3)='tennis'            manager(:e4)=:e1
hobby(:e3)='fishing'           hobby(:e4)='tennis'
hobby(:e3)='golf'              typingspeed(:e4)=1100
```

Figure 15: Extension of the object view of the company database (textual representation).

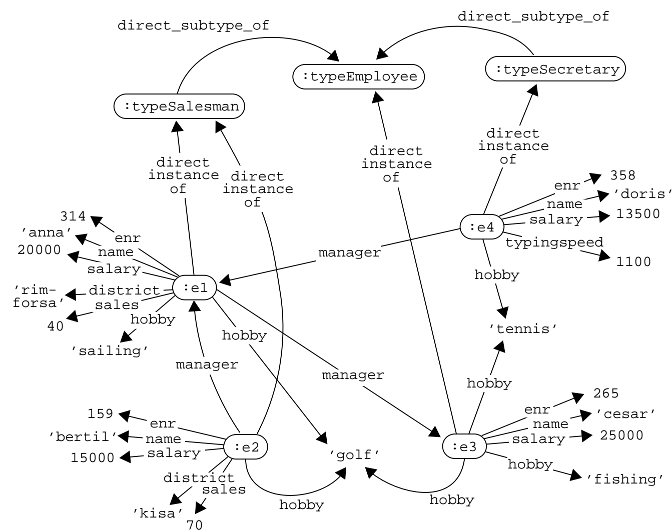


Figure 16: Extension of the object view of the company database (graphical representation).

8.2 Introduction to query processing

As an introduction, we will now give a brief overview of how an example query against the object view is processed. The details of this example will be given in later sections.

Consider the following query (in the AMOSQL language):

```
select s, salary(manager(s))
for each salesman s
where hobby(s)='golf'
```

A natural language formulation of this query would be something like: 'for each salesman that has golf as a hobby, retrieve that salesman together with the salary of his/her manager'.

If you look at the extension of the object view, you will see that there are two objects (:e1 and :e2) that are instances of the salesman type and that have golf as a hobby. Their managers are :e3 and :e1, respectively, and the salary of these objects are 25000 and 20000, respectively. Hence, the result of the query should be a bag of two tuples:

```
[<:e1,25000>, <:e2,20000>]
```

Since object identity is a concept of the object view and does not exist in the relational database, the query against the relational database will retrieve the employee number of the salesmen that satisfy the condition. The salesman objects that will be returned are the objects that correspond to these employee numbers.

The query will be translated into the following SQL query against the relational database:

```
select e1.enr, e2.salary
from employee e1 e2, emp_hobbies, salesman
where hobby='golf'
  and employee=e1.enr
  and e1.enr=salesman.enr
  and e1.manager=e2.enr
```

The result of the SQL query is a relation with two tuples:

```
{(314, 25000), (159, 20000)}
```

Let us assume that this is the first time a query retrieves these employees. In that case, the employee numbers 314 and 159 will not have any corresponding objects. Two new objects, :e1 and :e2, will be created and the mapping between these objects and the employee numbers 314 and 159 will be stored.

The new objects are returned together with the salary of their managers:

```
[<:e1,25000>, <:e2,20000>]
```

9 The Relational Data Model

This section gives a short overview of the relational data model. More thorough descriptions of the relational data model can be found in any basic database textbook, e.g. [20] [23] [62].

The relational data model was introduced by Codd [19] in 1970. A relational database is organized as a set of named tables (*relations*). The rows (*tuples*) of a table are not ordered. Each column (*attribute*) of a table has a name and an associated data type. Most existing relational databases require all relations to be in first normal form, i.e. no composite or multivalued attributes are allowed.

A *relational schema* defines the structure of a relation; the name of the relation and the name and data types of its attributes. A *relational database schema* is the set of all relational schemas. A relational schema is also called the *intension* of the relation. The actual data that is stored in a relation is called the *extension* of the relation.

A set of attributes which can be used to uniquely identify a tuple in a relation (no two tuples can have the same values for these attributes) is called a *superkey* for the relation. A *key* is a minimal superkey (all attributes of the key are needed to uniquely identify tuples; no attribute can be removed). *Candidate key* is a synonym to key. One of the candidate keys is chosen as the *primary key.* A set of attributes FK is a *foreign key* if: (a) they have the same domain as the primary key PK of some relation R, and (b) all values of FK must also exist in PK.

Figure 17 shows the relational database schema for the company example. The data types for the attributes are not included.

Figure 18 shows the extension of the relation *employee* in the company example.

Constraints can be specified on relations to guarantee the integrity of data. *Key constraints* are used to guarantee that candidate key values are unique for all tuples. *Referential integrity constraints* guarantee that all the values of a foreign key exist as values of the corresponding primary key.

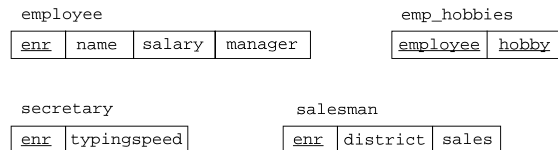


Figure 17: Relational database schema (intension) for the company example. Underlined attributes are part of the primary key for the relation.

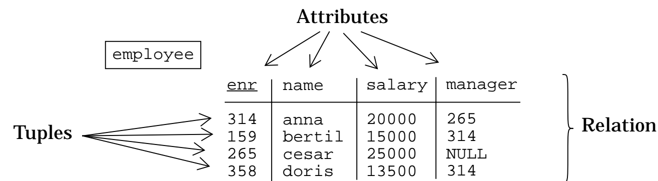


Figure 18: Extension of the relation *employee* in the company example.

An *inclusion dependency* R[X] ⊆ S[Y] between a set of attributes X of a relation R and a set of attributes Y of a relation S means that the values of X must also exist as values of Y. R and S need not be different relations. The following inclusion dependencies hold in the example in figure 17:

```
emp_hobbies[employee] ⊆ employee[enr]
secretary[enr] ⊆ employee[enr]
salesman[enr] ⊆ employee[enr]
employee[manager] ⊆ employee[enr]
```

The two most important ways to manipulate data in a relational database are through the relational algebra or through the declarative language SQL (which is based on the relational calculus). The relational algebra is usually used as an internal representation in DBMSs. SQL is the de facto standard language for end-users and application programming interfaces.

10 Object-Oriented Data Models

Unlike the case with relational databases, there is no single accepted data model for object-oriented databases. Different object-oriented DBMSs all use slightly different data models. There are, however, some characteristics that are common to all object-oriented data models. This section gives an overview of these characteristics. Note the close correspondence to the semantic modelling constructs discussed in section 5.1.

Objects. The basic modelling construct is the object. Objects are used to model entities in the domain of interest. To varying extents in different object models, *everything* is an object, including other modelling constructs such as types and methods.

State. Each object has a set of attributes. This may be referred to as the structural part of the object. The value of an attribute may be another object in which case the attribute models a relationship between objects. The values of the attributes define the *state* of the object.

Behaviour. The behaviour of an object is defined by the operations (methods) which can be carried out on the object.

Encapsulation. In object-oriented programming languages, encapsulation states that the only way to access an object is through its methods. The structural part of the object is not directly accessible. In object-oriented databases, this strict notion of encapsulation is often violated. Many object-oriented databases allow direct access to the structural part of objects.

Object identity. Each object has a unique, immutable identifier which can always be used to refer to it. This means that objects have an existence which is independent of the values of their attributes, something which is different from value-based data models such as the relational data model.[32]

Types and instances. Objects are categorized by types. All instances of a type have the same attributes and behaviour.

Generalization/specialization. Types are organized as subtypes/supertypes to each other. The subtype inherits the attributes and behaviour of the supertype and may have additional attributes and methods that the supertype does not have.

Complex objects. Aggregation (see section 5.1) is supported implicitly since objects have attributes which may take other objects as values. It is

32. See footnote 10 on page 24.

also possible to create complex objects of this kind explicitly by applying constructors such as *tuple* or *record* on existing objects. Grouping is supported by constructors such as *set* and *bag*. Order between objects is supported by constructors such as *list* and *array.*

In the AMOS data model, *functions* are used to model both properties of objects, relationships between objects, and operations on objects.

A consequence of the lack of a standard object-oriented data model is that there is no standard query language for object-oriented databases. Many projects use a query language with a syntax similar to SQL, the reason for this being that SQL is such a wide-spread language.

The ODMG standard [16] is the first attempt from object-oriented DBMS vendors to define a standard for object-oriented data models and query languages.

A current trend is to extend relational database systems with object-oriented concepts. These systems are usually called object-relational database systems. The Montage DBMS is the first commercially available system of this kind. When the relational data model is extended with object-oriented concepts, the SQL language has to be extended accordingly. The next revision of the ANSI SQL standard, SQL3, is expected to contain such extensions [7].

11 Mapping Between the Relational and an Object-Oriented Data Model

This section discusses the relationship between the relational data model and object-oriented data models. We will use the AMOS data model as an example of an object-oriented data model.

Often, there is a close correspondence between types in the AMOS data model and relations in the relational data model, between objects and tuples, and between functions and attributes. Consider for example the employee relation in the company database. The corresponding AMOS schema has a type employee, and there are four instances of that type - one for each tuple of the employee relation. The attributes enr, name, salary, and manager in the employee relation each correspond to a function defined on the employee type. This is all illustrated in figure 19.

In reality, there is no one-to-one correspondence between relational modelling constructs and AMOS modelling constructs. There are always different ways to map between constructs in one of the data models to constructs in the other. In particular, there are some concepts in object-oriented data models for which there are no corresponding concepts in the relational data model.

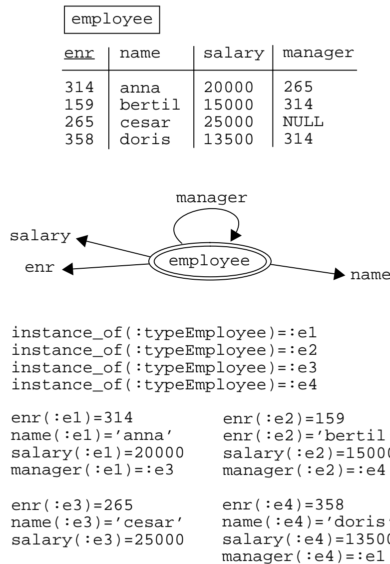
Complex objects are not supported by the relational data model, i.e. no multivalued or composite attributes are allowed. Multivalued functions in AMOS require separate relations in a relational database. Composite attributes must be split up.

Generalization/specialization in object-oriented data models has no correspondence in the relational data model. When subtype/supertype relationships exist between objects in the domain which is modelled, this is only represented implicitly in relational databases.

The concept of object identity is not present in the relational data model. However, this is not a problem at the schema level and we will come back to this in section 13.

The mapping from object-oriented to relational schemas is well understood, since it is a central part of many database design methodologies [23].[33] If the methodology is followed, the resulting relational schema automatically

33. The database schema is initially expressed in a semantically rich data model (often an EER data model) and is transformed to a relational schema using some algorithm.



employee

enr	name	salary	manager
314	anna	20000	265
159	bertil	15000	314
265	cesar	25000	NULL
358	doris	13500	314


```
instance_of(:typeEmployee)=:e1
instance_of(:typeEmployee)=:e2
instance_of(:typeEmployee)=:e3
instance_of(:typeEmployee)=:e4

enr(:e1)=314            enr(:e2)=159
name(:e1)='anna'        enr(:e2)='bertil'
salary(:e1)=20000       salary(:e2)=15000
manager(:e1)=:e3        manager(:e2)=:e4

enr(:e3)=265            enr(:e4)=358
name(:e3)='cesar'       name(:e4)='doris'
salary(:e3)=25000       salary(:e4)=13500
                        manager(:e4)=:e1
```

Figure 19: A relational schema with its extension (top), the corresponding AMOS schema (middle), and the extension of the AMOS schema (bottom).

has the properties of 'good' data design.[34]

In an ideal world, all relational databases would have schemas of this kind. Unfortunately this is not the case - there are many different ways to model some information and people will not always choose the 'best'. This may be due to bad database design, but it may also be due to special requirements on the database, such as performance. In particular, there are many ways to model subtype/supertype relationships in relational schemas. This is discussed in section 11.1.

Section 11.2 discusses ways to identify semantic modelling constructs in relational database schemas, and how to transform a relational database schema to an object-oriented schema.

To make the mapping from relational to object-oriented structures easier, we have defined a normal form for representing subtype/supertype relationships in relational schemas. If the relational schema is not in the normal form, a relational view is defined which *is*. This is the subject of section 11.3.

34. Natural, easy to understand, no redundancy.

11.1 From object-oriented to relational

As mentioned above, the process of mapping a schema in a semantically rich data model to a relational database schema is often performed during database design. We refer to [23] for a complete algorithm for mapping an EER schema to a relational database schema. A general observation is that some semantics which were explicitly represented in the EER/OO schema are only implicitly represented in the relational database schema. The only thing we will discuss in more detail in this section is different ways to represent subtype/supertype relationships in the relational data model.

Figure 20 shows four alternative mappings from an AMOS schema to a relational database schema (adapted from [23]).

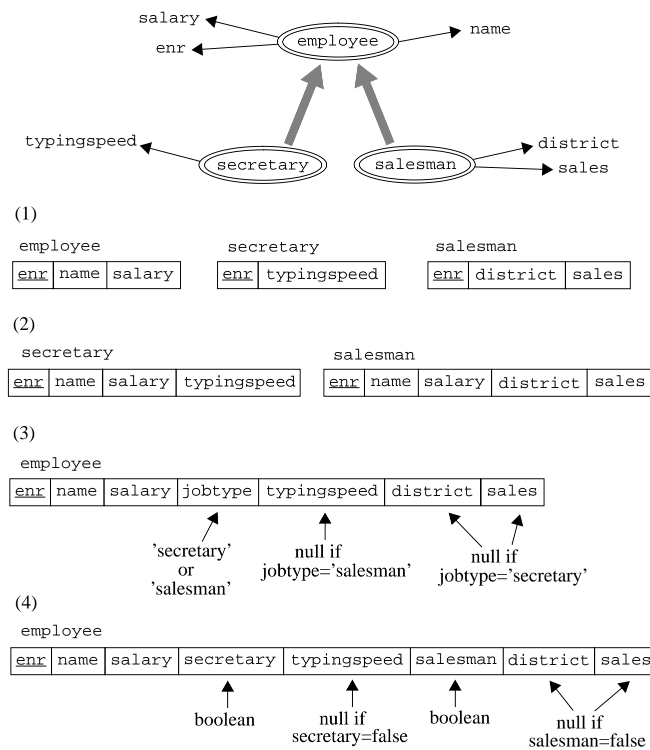


Figure 20: Representing subtype/supertype relationships in relational schemas. Four alternative mappings from the AMOS schema (top) to relational schemas are given.

Alternatives (1) and (2) are probably the most common. Note that the AMOS schema is a subset of the company example used elsewhere in the thesis.

In alternative (1) all types get their own relation. This representation can be used for all kinds of specialization (disjoint/overlapping and total/partial).[35]

In alternative (2) there is no relation for the supertype. The attributes (functions) of the supertype are duplicated in all the relations representing the subtypes. This can not be used for partial specialization.

Alternative (3) has one relation for all types. The jobtype attribute specifies the type of the employee (secretary or salesman).[36] The relation schema contains all the attributes of all the subtypes. If an attribute is not applicable (e.g. typingspeed for salesmen) it is given the value null. This representation can not be used for overlapping specialization.

Alternative (4) also has a single relation for all types. For each subtype there is a boolean-valued attribute which specifies whether the employee is an instance of that type or not. The relation schema contains all the attributes of all the subtypes. If an attribute is not applicable it is given the value null. This representation can be used for all kinds of specialization.

11.2 From relational to object-oriented

Quite a lot of work has been done on how to identify semantic modelling constructs in relational database schemas [4] [34] [49] [51] [64]. Most of the methodologies in these papers are based on classifying relations based on their primary keys and inclusion dependencies. The methodologies then automatically transfer the relational schema to an extended entity-relationship (EER) or object-oriented (OO) schema.

As an example, we will show how the relational database schema for the company database is mapped to an OO schema using the methodology in [4]. The relational database schema is shown in figure 13 on page 47. The inclusion dependencies are given on page 52.

The relations employee, secretary, and salesman all have a primary key consisting of a single attribute, and they are therefore classified as relations of category A.[37] The emp_hobbies relation has a composite primary key,

35. See page 25.

36. Or null, if the employee is neither a secretary nor a salesman ('just' an employee).

37. Actually, the methodology classifies *equivalence classes* of relations, rather than single relations. This makes it capable of handling vertical partitioning of relations. None of the relations in our example are vertically partitioned and for ease of discussion we therefore only deal with single relations.
The methodology distinguishes between six different categories of equivalence classes of relations.

the primary key is *not* a foreign key, and some, but not all, of the primary key attributes are foreign keys. The emp_hobbies relation is therefore classified as a category E relation.

For each relation of category A, a type is created in the OO schema. This is illustrated in figure 21(a). For each attribute of the relations, a function is defined on the corresponding type. If the attribute is not a foreign key, then the return type of the function is the literal type corresponding to the domain of the attribute. If the attribute *is* a foreign key, then the return type is the type corresponding to the relation which the foreign key references. This is illustrated in figure 21(b).

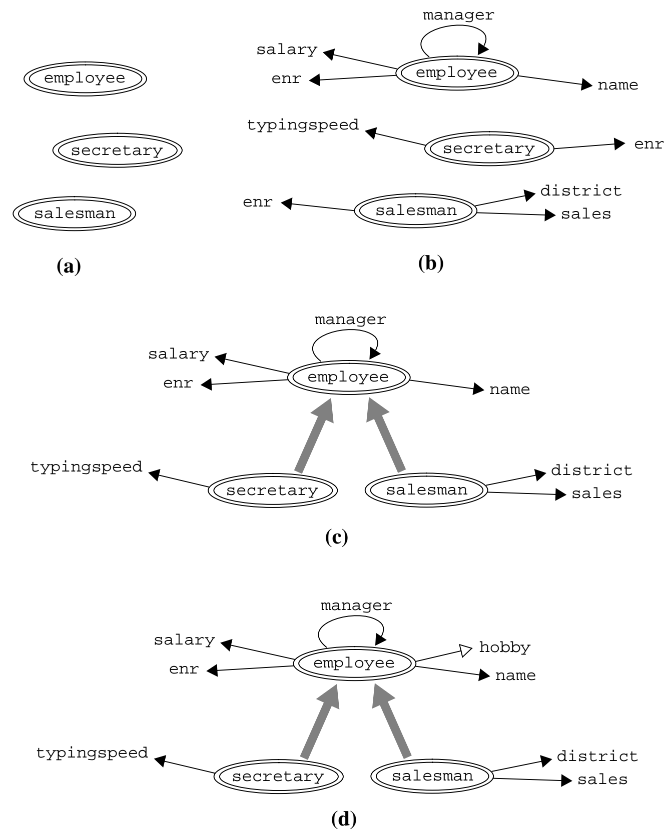


Figure 21: Mapping the relational database schema of the company database to an object-oriented schema, according to the methodology in [4].

Since there is an inclusion dependency where the left side is the primary key of the secretary relation and the right side is the primary key of the employee relation, the secretary type is declared as a subtype of the employee type. Similarly, the salesman type is a subtype of the employee type. This is illustrated in figure 21(c).

Relations of category E are mapped to multivalued functions in the OO schema. Since the employee attribute of the emp_hobbies relation is a foreign key which references the relation employee, a multivalued function is defined on the employee type. The range of the function is the literal type corresponding to the domain of the non-foreign key attribute (hobby) of the emp_hobbies relation. This gives us the final OO schema, which is illustrated in figure 21(d).

We believe that a completely automatic transformation of relational database schemas to EER/OO schemas is not possible. Even if all primary keys and inclusion dependencies for a relational database schema are given, it is possible to map this to different EER/OO schemas. An example of this is that some relations in relational schemas may be mapped to either entities or relationships in an EER schema. The mapping process must always be guided by a DBA. The methodologies may be very helpful as tools during schema mapping, though.

11.3 A normal form for representing subtype/supertype relationships in relational databases

As could be seen in section 11.1, there are many different ways to represent subtype/supertype relationships in a relational database schema. We have defined a normal form, SSNF (Subtype/Supertype Normal Form), for representing these kinds of structures in relational database schemas. If a relational database schema is in SSNF, the mapping between the relational database and the object view is greatly simplified. When an object view is defined over a relational database that is not in SSNF, the first step is to define a relational view that *is* in SSNF.

It is not possible to determine whether a relational database schema is in SSNF by looking at it in isolation. A relational database schema is (or is not) in SSNF *with respect to an EER/OO schema* (or some other schema which has the notion of subtypes and supertypes).

We define SSNF as follows:

Let OS be an EER/OO schema and RS the corresponding relational database schema. RS is in SSNF if:

For each type T in OS there exists a relation R in RS such that there is a one-to-one mapping between instances of T and tuples in R.

(End of definition.)

In figure 20 on page 57, schema (1) is in SSNF whereas schemas (2)-(4) are not.

The main benefit of having the relational schema in SSNF is that the object view mechanism does not need to handle all possible cases of relational database schemas. It only needs to handle one case.

The reason for choosing the particular representation we have chosen is that it simplifies management of the instance-of relationship. To check whether an object is an instance of a particular type, or to retrieve all instances of a particular type, it suffices to examine a single relation.[38] This should become clear in section 13.

When an object view over a relational database that is not in SSNF is going to be created, the developer must first create a relational view that *is* in SSNF.

The following SQL statements can be used to define an SSNF view over schema (2) in figure 20:

```
create view employee* as
  (select enr, name, salary from secretary)
  union
  (select enr, name, salary from salesman)

create view secretary* as
  select enr, typingspeed from secretary

create view salesman* as
  select enr, district, sales from salesman
```

An SSNF view over schema (3) in figure 20 could be defined with the following SQL statements:

```
create view employee* as
  select enr, name, salary from employee

create view secretary* as
  select enr, typingspeed
  from employee
  where jobtype='secretary'

create view salesman* as
  select enr, district, sales
  from employee
  where jobtype='salesman'
```

38. The *pivot* relation plays a similar role in the PENGUIN system [41]. In PENGUIN, the extension of the object schema corresponding to a relational database schema is materialized once and for all.

And to define an SSNF view over schema (4) in figure 20, the following SQL statements could be used:

```
create view employee* as
  select enr, name, salary from employee

create view secretary* as
  select enr, typingspeed
  from employee
  where secretary=true

create view salesman* as
  select enr, district, sales
  from employee
  where salesman=true
```

Note that it is not possible to create SSNF views for all kinds of relational database schemas. For example, the SSNF view over schema (3) in figure 20 could only be created because we assumed that the domain of the jobtype attribute was fixed. Suppose that this assumption could not be made, i.e. that the domain of the jobtype attribute was character strings in general rather than the two specific character strings 'secretary' and 'salesman'. In that case, the SSNF view should have one relation for each distinct value that occurred in the jobtype column. In other words, the number of relations in the view would be dependent on the state of the underlying database. Such views are not possible to create in current relational database systems. More general view mechanisms for relational databases are discussed in [39] and [45].

12 Query Processing in Object-Oriented DBMSs

Access to object-oriented databases [5] [8] [15] can be performed in two different ways; by following pointers or by declarative queries. An example of access by following pointers is the following query:

```
manager(manager(:e2));
```

The user starts with a handle (pointer) to the object :e2. Retrieving the manager (:e1) of :e2 can be seen as following a pointer from :e2 to :e1 (see figure 16 on page 49). The manager of :e1 is retrieved by following a pointer from :e1 to :e3.

The following is an example of a declarative query:

```
select name, sales
for each salesman s
where hobby(s)='golf'
  and district(s)='kisa'
```

An object-oriented DBMS should support efficient execution of both kinds of queries. It is the second kind of query which makes query optimization an important issue. There are many different ways to process a query like this, and the costs of the alternative execution strategies may be very different. The task of the query optimizer is to find the most efficient execution strategy.

We will use AMOS to illustrate query processing in object-oriented DBMSs.

Most object-oriented DBMSs use expressions in an object algebra as the internal representation of query plans. Unlike the case of relational algebra, there is no standard object algebra. Different object algebras have been proposed by different authors [21] [60].

AMOS uses a logical representation for query plans. Since the algebraic approach is so dominant, we will take some time to describe the relationship between these two approaches. This will be done in section 12.1.

Section 12.2 discusses heuristic and cost-based query optimization techniques.

Finally, section 12.3 gives an overview of query processing in AMOS.

12.1 Internal representation of query plans

When a query is processed it is first translated into an internal representation which is easy to optimize.

The most common internal representation of queries is *algebraic expressions*. For example, in relational database systems, SQL queries are usually translated to equivalent expressions in *relational algebra*.

Another approach is to use a *logical representation* of query plans. The logical language used is usually not a general-purpose programming language, but rather a subset which makes it more suitable as a database language. An example of this is the *Datalog* language [62] [17], which is a subset of Prolog. The main difference between Prolog and Datalog is that Datalog predicates can not have function symbols as arguments. All arguments are constants or variables.

We use the logical approach in the AMOS system.

The expressive power of relational algebra and Datalog is very similar[39], which leads one to suspect that the difference between the two approaches is mainly syntactical. We will show that this *is* the case. This has the pleasant consequence that the results in this thesis are equally applicable to systems where algebraic expressions are used as the internal representation.

We will use relational algebra and Datalog to illustrate the relationship between the algebraic and the logical approach. Consider the following SQL query:

```
select name, sales
from employee, salesman, emp_hobbies
where employee.enr=salesman.enr
  and employee.enr=emp_hobbies.enr
  and hobby='golf'
  and district='kisa'
```

If Datalog is used as the internal representation, the query will be translated to the following Datalog rule, where temp is a temporary predicate which is used to hold the result variables:

```
temp(NAME,SLS)   <- employee(ENR,NAME,_,_) &         (1)
                    emp_hobbies(ENR,'golf') &        (2)
                    salesman(ENR,'kisa',SLS)         (3)
```

There are six possible orderings of the Datalog predicates. Each of these orderings corresponds to an execution strategy. This is illustrated in figure 22.

39. Relational algebra without negation and Datalog without recursion have the same expressive power. Datalog is strictly more expressive than relational algebra without negation. Relational algebra is strictly more expressive than Datalog without recursion [62] [17].

```
(1)(2)(3)     temp(NAME,SLS) <- employee(ENR,NAME,_,_) &
                                emp_hobbies(ENR,'golf') &
                                salesman(ENR,'kisa',SLS)

(1)(3)(2)     temp(NAME,SLS) <- employee(ENR,NAME,_,_) &
                                salesman(ENR,'kisa',SLS) &
                                emp_hobbies(ENR,'golf')

(2)(1)(3)     temp(NAME,SLS) <- emp_hobbies(ENR,'golf') &
                                employee(ENR,NAME,_,_) &
                                salesman(ENR,'kisa',SLS)

(2)(3)(1)     temp(NAME,SLS) <- emp_hobbies(ENR,'golf') &
                                salesman(ENR,'kisa',SLS) &
                                employee(ENR,NAME,_,_)

(3)(1)(2)     temp(NAME,SLS) <- salesman(ENR,'kisa',SLS) &
                                employee(ENR,NAME,_,_) &
                                emp_hobbies(ENR,'golf')

(3)(2)(1)     temp(NAME,SLS) <- salesman(ENR,'kisa',SLS) &
                                emp_hobbies(ENR,'golf') &
                                employee(ENR,NAME,_,_)
```

Figure 22: Possible execution plans for a Datalog rule with three subgoals.

There are two main alternatives for join processing in relational databases; *pipelining* and *materialized intermediate relations* [38]. Pipelined join processing is a way to avoid creation of large intermediate relations. Sometimes, however, materialization of intermediate results can improve performance. A combination of the two approaches is often advantageous.

We will use execution plan (2)(3)(1) from figure 22 to illustrate pipelined join processing and the possible benefits of materializing intermediate relations. A join processing strategy without materializations will be referred to as *pure pipelining*.

The query processor starts by searching the emp_hobbies relation for tuples where hobby='golf'. As soon as one such tuple is found, the query processor moves on to the salesman relation. There, it searches for tuples where the enr value is equal to the enr value of the emp_hobbies tuple. Let us assume that there is exactly one such tuple. As soon as this tuple is found, the query processor moves on to the employee relation. Again, it searches for tuples where the enr value is equal to the enr value of the emp_hobbies tuple (and of the salesman tuple). There will be exactly one such tuple, and the name value of that tuple, together with the sales value of the salesman tuple, is returned as a result of the query. The query processor then *backtracks* and retrieves the next tuple in the emp_hobbies relation where hobby='golf'. Again, it searches the salesman relation for tuples where the enr value is equal to the enr value of the current emp_hobbies tuple. And so on. No intermediate relations are created.

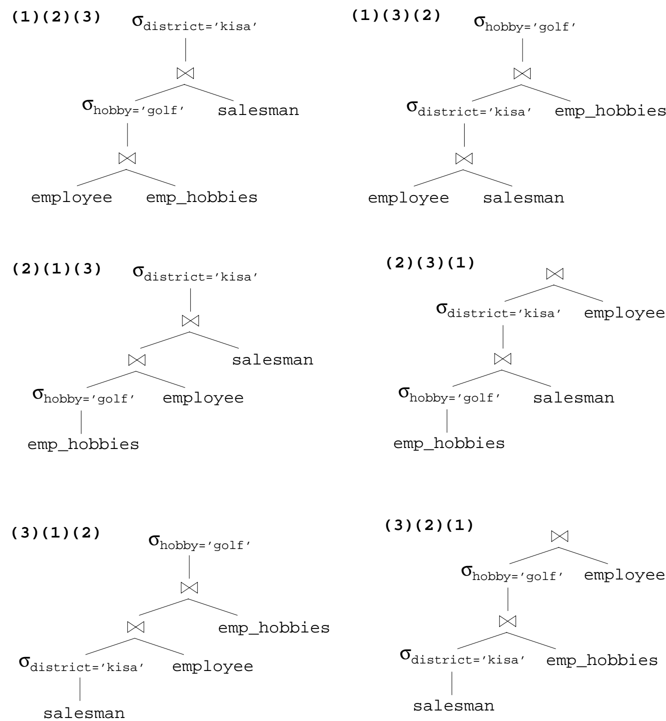


Figure 23: Relational algebra expressions corresponding to the different Datalog execution plans in figure 22.

Suppose now that there is no index on the enr attribute of the salesman relation. This means that every time the query processor searches the salesman table for a particular value of enr (which it does once for each tuple in emp_hobbies where hobby='golf'), it will have to scan the entire relation. When a matching tuple is found, the query processor checks that district='kisa'. A better strategy would be to create an intermediate relation which is the subset of the salesman relation where district='kisa'. This would reduce the cardinality of the relation that has to be searched repeatedly.[40]

40. Another alternative would be to perform a 'hash join' [23].

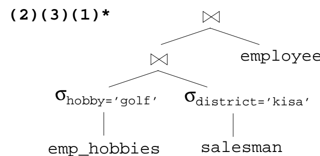


Figure 24: Example execution plan represented in relational algebra. The select operation on the salesman relation results in a materialized intermediate relation.

```
(2)(3)(1)*    temp(NAME,SLS) <- (emp_hobbies(ENR,'golf') ∧
                                  salesman(ENR,'kisa',SLS)) &
                                 employee(ENR,NAME,_,_)
```

Figure 25: Datalog execution plan corresponding to the relational algebra expression in figure 24.

Let us assume that the '&' symbol in the execution plans means 'pipelined join processing'. In that case, the six Datalog execution plans in figure 22 correspond to the relational algebra expressions illustrated in figure 23. The expressions are represented as tree structures. The project operations are not included; only the join and the select operations are. When pure pipelining is used, the relational algebra tree must always be a binary tree where the right branch of every node is a leaf.

Figure 24 illustrates the relational algebra tree for the alternative strategy for the execution plan (2)(3)(1) discussed above. Here, an intermediate relation is created which contains the tuples of the salesman relation where district='kisa'.

With the Datalog notation we have used so far, there is no corresponding Datalog execution plan for the relational algebra expression in figure 24. One possible way to represent join processing with materialization in Datalog execution plans would be to add a new 'and' symbol. Let us call this symbol '∧'. When a '∧' symbol is encountered in an execution plan it would mean 'materialize the relation corresponding to the right side of this expression. The Datalog execution plan corresponding to the relational algebra expression in figure 24 would then look as in figure 25.

12.2 Heuristic vs cost-based optimization

There are two main alternative techniques for finding an optimal execution plan; *heuristic rules* and *cost estimation* [23].

When heuristic rules are used, the optimizer maintains a set of equivalence-preserving transformation rules for query plans. The heuristic rules state

which of two equivalent query plans is usually the most efficient. An important heuristic rule for relational algebra expressions is 'selection pushing', which states that it is almost always better to perform select operations before joins. A heuristic rule for Datalog programs is 'bound-is-easier' [62], which states that it is usually better to order the subgoals so that the number of bound arguments during execution is as high as possible.

When the cost estimation approach is used, the optimizer uses information about access paths, communication costs, processing costs, etc., to systematically estimate the cost for the different execution alternatives. The execution plan with the lowest cost is then chosen. Provided that the input information is correct, this approach always yields the optimal execution strategy, but at a much higher cost than the heuristic approach. If the search space is very large (more than 10 joins in the relational database case), it is totally infeasible to estimate the cost of all possible execution strategies. Randomized optimization methods [33] is one way to handle large search spaces.

The AMOS query optimizer uses the cost estimation approach and provides three alternatives for investigation of the search space: (1) exhaustive search, (2) limited search space through greedy heuristics [47], and (3) randomized optimization. A detailed description of the AMOS cost model can be found in [47].

12.3 Query processing in AMOS

This section gives an overview of query processing in AMOS. We will only give a cursory introduction, the details can be found in [47].

The different phases of query processing in AMOS are given in figure 26.

AMOS uses the logical language ObjectLog as the internal representation of queries. Query optimization, then, is the process of finding the optimal ordering of the ObjectLog subgoals. ObjectLog can be characterized as 'object-oriented Datalog'. ObjectLog predicates are typed and can be overloaded on all arguments, they can take object identifiers as arguments, and they are first-class objects. More on the ObjectLog language can be found in [47].

To illustrate the query processing steps, we will use the following example query:

```
select s, doublesalary(manager(s))
for each salesman s
where hobby(s)='golf'
```

For the sake of this example, we assume that doublesalary is defined as a derived function:

```
create function doublesalary(employee e)->integer ds as
  select salary(e)*2
```

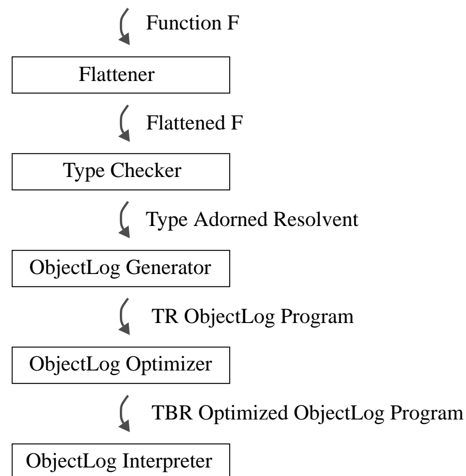


Figure 26: Query processing steps in AMOS.

The query processing steps are identical for ad hoc queries and for creation of derived functions. Therefore, when an ad hoc query is sent to the query processor, it first creates a temporary function (derived) whose definition is the ad hoc query. In our example, that function would be defined as follows:

```
create function temp()-><salesman,integer> as
  select s, doublesalary(manager(s))
  where hobby(s)='golf'
```

Flattener

The function is first run through the Flattener. Nested function calls are removed by introduction of intermediary variables. The flattened function is:[41]

```
create function temp()-><salesman s,integer ds> as
  select s, ds
  for each employee m, charstring h
  where ds=doublesalary(m)
    and m=manager(s)
    and h=hobby(s)
    and h='golf'
```

41. The temporary variables will of course not have names like s, ds, m and h but rather _v1, _v2, _v3 and _v4 etc. We use names like s, ds, m and h to make the examples easier to follow.

Type Checker

Next, the function passes the Type Checker. Here, the resolvents of the functions are identified. Each function call is replaced with a call to one of its type-adorned (TA) resolvents. In the case of late binding, function name resolution is postponed until run-time.

The type checker also adds dynamic type checks if the function calls are not enough to guarantee correct type membership. In this example, the type membership of s needs to be checked since all function calls involving s are defined on the more general type employee. Without the type check, s could be bound to employee objects which are not salesmen.

```
create function temp_->salesman,integer()-><s,ds> as
  select s,ds
  where ds=doublesalary_employee->integer(m)
    and m=manager_employee->employee(s)
    and h=hobby_employee->charstring(s)
    and h='golf'
    and typesof_object->type(s)=:typeSalesman
```

Functions and Predicates

The next step of query processing is to transform the TA resolvent function into the internal representation of queries; ObjectLog. Each TA resolvent function has a corresponding type resolved (TR) ObjectLog predicate. This ObjectLog predicate may be stored directly in the database (i.e. a fact) or it may be defined in terms of other predicates (i.e. a rule).

Stored functions become TR facts. For example, the TA resolvent

```
manager_employee->employee
```

has the following corresponding TR fact:

```
manager_employee,employee(E,M)
```

Derived functions become TR rules. For example, the TA resolvent

```
doublesalary_employee->integer
```

has the following corresponding TR rule:

```
doublesalary_employee,integer(E,DS)
                 <- salary_employee,integer(E,S) &
                    times_integer,integer,integer(S,2,DS)
```

ObjectLog Generator

The transformation from TA resolvent functions to ObjectLog is performed by the ObjectLog Generator.

The temp function is a derived function and is therefore transformed to the following TR rule:

$\texttt{temp}_{salesman,integer}$(S,DS)
    <- $\texttt{salary}_{employee,integer}$(M,SAL) &
       $\texttt{times}_{integer,integer,integer}$(SAL,2,DS) &
       $\texttt{manager}_{employee,employee}$(S,M) &
       $\texttt{hobby}_{employee,charstring}$(S,'golf') &
       $\texttt{typesof}_{object,type}$(S,:typeSalesman)

ObjectLog Optimizer

The ObjectLog Optimizer decides the execution order of the subgoals of the TR rule. The output of the optimizer is a type and binding pattern resolved (TBR) rule. ObjectLog predicates are overloaded both on type and on binding pattern. For example, there are four TBR predicates for the $\texttt{typesof}_{object,type}$(OBJ,TP) predicate:

$\texttt{typesof}^{ff}_{object,type}$(OBJ,TP)
$\texttt{typesof}^{fb}_{object,type}$(OBJ,TP)
$\texttt{typesof}^{bf}_{object,type}$(OBJ,TP)
$\texttt{typesof}^{bb}_{object,type}$(OBJ,TP)

An 'f' in the binding pattern means that the corresponding variable is *free* at the time of execution. A 'b' in the binding pattern means that the corresponding variable is *bound* at the time of execution.

Which execution order the optimizer chooses depends on the cost to execute the different TBR predicates and on the cardinality of the output (fanout) from them [47]. Estimation of the cost and fanout for stored predicates is based on the cardinality of the predicates and on what indexes are available. Estimation of the cost and fanout for foreign predicates is performed by special cost hint functions. The cost hint function for a foreign predicate is supplied to the optimizer by the implementor of the predicate.

Based on what arguments are bound at execution time, each subgoal is replaced with one of its corresponding TBR predicates. The following TBR rule seems to be a good candidate for optimal execution order:

$\texttt{temp}^{ff}_{salesman,integer}$(S,DS)
    <- $\texttt{hobby}^{fb}_{employee,charstring}$(S,'golf') &
       $\texttt{typesof}^{bb}_{object,type}$(S,:typeSalesman) &
       $\texttt{manager}^{bf}_{employee,employee}$(S,M) &
       $\texttt{salary}^{bf}_{employee,integer}$(M,SAL) &
       $\texttt{times}^{bbf}_{integer,integer,integer}$(SAL,2,DS)

ObjectLog Interpreter

The ObjectLog Interpreter executes the TBR program. The interpreter uses pure pipelining as the join processing strategy (no intermediate results are materialized).

13 Object Views of Relational Data

The term *Translator* was introduced in section 6 to denote the software that implements the mapping between a local schema and a component schema. It provides a view of a component database where the data models of the view and the component database may be different.

When the term Translator is used in this section, unless otherwise stated, it will refer to the specific kind of Translator where the local schema is expressed in the relational data model and the component schema is expressed in the AMOS data model.

A Translator provides the functionality of an AMOS DBMS augmented with the notions of *mapped types* and *mapped objects*.

A mapped type is a type for which the extension is defined in terms of the state of an external database. In our case the external database is relational, and the extension of a mapped type is defined so that there is a one-to-one mapping between instances of the mapped type and tuples in some relation in the external database.[42] The instances of mapped types are called mapped objects.[43]

Figure 27 shows the subtype/supertype graph for Translators. Mapped types are subtypes to the type usertypeobject and are instances of the type mappedtype.

Figure 28 shows how mapped types fit into the subtype/supertype graph. We use the delimitation that mapped types can not have more than one direct supertype (no multiple inheritance).

We will use the term *most general mapped type* (*mgmt* for short) for mapped types that are direct subtypes to usertypeobject. In figure 28, the types MT_1 and MT_6 are most general mapped types. In the company example, the type employee is the only mgmt.

42. Recall the SSNF normal form for relational databases discussed in section 11.3.

43. The notions of mapped types and mapped objects are similar to what is called *virtual classes* and *imaginary objects* in [1]. The main difference is that the object view in [1] is defined over an object-oriented database, rather than over a relational database. The extension of a virtual class in [1] is defined in terms of the state of the object-oriented database.

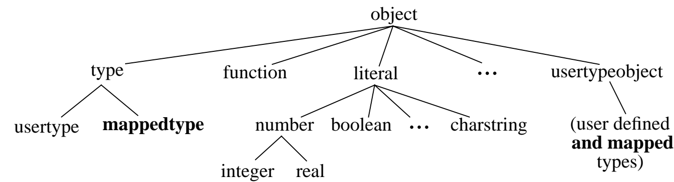


Figure 27: Subtype/supertype graph for Translators (compare with figure 8 on page 33).

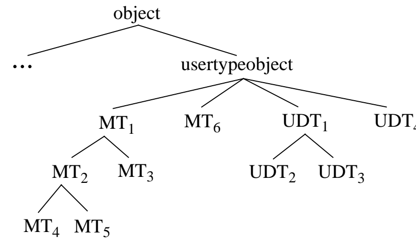


Figure 28: Mapped Types ($MT_{1\text{-}6}$) and User Defined Types ($UDT_{1\text{-}4}$) in the subtype/supertype graph for Translators.

We will also use a function called *mgmt*. If a mapped type T is given as argument to the function, it returns the supertype to T which is a mgmt. If a mapped object (OBJ) is given as argument to the function, it returns mgmt(T_{OBJ}), where T_{OBJ} is the mapped type that OBJ is a direct instance of.

The following are examples of the results of applying the *mgmt* function to types and objects in the company example.

```
mgmt(:typeSalesman) = :typeEmployee
mgmt(:typeEmployee) = :typeEmployee
mgmt(:e4) = mgmt(:typeSecretary) = :typeEmployee
```

The examples in this section presuppose an understanding of the AMOS data model (section 6.2), the AMOSQL query language (section 6.3), and query processing techniques in AMOS (section 12.3). The reader should be familiar with concepts like type-resolved (TR) predicates and type-and-binding-pattern resolved (TBR) predicates, and their role during query processing.

We will not discuss how the object-oriented schema is defined in terms of the relational schema. The mapping between the relational database and the object view should be defined using a declarative object view definition

language (OVDL). In the current prototype, the mapping is established by handcoding the effects of imagined OVDL statements. See [4] for an example of what such a view definition language can look like.

The rest of this section is organized as follows.

Section 13.1 shows how relational database access is represented in query plans in the Translator. A recurring theme throughout this section (13) is the importance of representing all relational database access explicitly in query plans. No predicate should have relational database access embedded in the code which implements it.

Section 13.2 deals with the concept of object identity. It shows how object identity can be provided in the object view even though there is no such concept in the relational database.

Section 13.3 discusses the instance-of relationship in object views. In an object view of a relational database, the relationship between objects and types depends on the state of the relational database.

Finally, section 13.4 discusses query optimization techniques in object views of relational data.

13.1 Representing relational database access in query plans

In most cases it is advantageous to translate AMOSQL queries to as few and as large SQL queries as possible. A naive translation method would lead to a large number of small queries against the relational database. This would result in unnecessary communication between the Translator and the relational database.

To make it possible for the optimizer to generate optimal execution plans, it is essential that all relational database access is *represented explicitly in query plans*. We use a special kind of functions, *r-functions*, and a special kind of predicates, *r-predicates*, for this.

To illustrate the necessity of this, we will start in section 13.1.1 by showing what the effects of a naive strategy would be. Section 13.1.2 discusses the use of multi-way foreign functions, which gives more flexibility but which is not enough to generate optimal execution plans. Section 13.1.3 introduces r-functions and r-predicates and shows how relational database access is represented in AMOS query plans.

13.1.1 Naive strategy

Consider the following function in the object view:

```
salary(employee e)->integer sal
```

When this function is used in a query, the Translator must access the employee relation in the relational database. This is where the information about salaries of employees is stored.

The simplest (and most naive) way would be to define the function as a foreign function with the following implementation (in pseudo code):[44]

```
Let X be the employee number of E. Execute the SQL query
'select salary from employee where enr=X'. Return the
salary that is returned.
```

This would make it possible to process a query like:

```
select salary(:e1)
```

The fact that the salary of :e1 is stored in a relational database would be transparent to users of the view. However, this simple approach would not allow much query optimization (as will be discussed later) and the function could only be used efficiently in the forward direction, i.e. when the employee is known but the salary is not. For example, to find out which employees earn 15000, it would be necessary to retrieve the salary for each employee and then check whether it is equal to 15000. With *n* employees in

44. The mapping between employees and employee numbers will be discussed in section 13.2.

the database, *n* SQL queries would be sent to the relational database.

13.1.2 Multi-way foreign functions

A more flexible, but still naive, way is provided with multi-way foreign functions [47]. There would be four different TBR predicates for the function. The following is an informal description of their semantics:

1) Neither the employee nor the salary is known, i.e:

$\texttt{salary}^{ff}_{employee,integer}\texttt{(E,SAL)}$

Execute the SQL query 'select enr, salary from employee'. For each result tuple <e#,s>, let (e,s) be a result binding to (E,SAL), where e is the employee object which corresponds to e#.[45]

2) The salary is known, but the employee is not, i.e:

$\texttt{salary}^{fb}_{employee,integer}\texttt{(E,SAL)}$

Execute the SQL query 'select enr from employee where salary=SAL'. For each result tuple <e#>, let (e,SAL) be a result binding to (E,SAL), where e is the employee object which corresponds to e#.

3) The employee is known, but the salary is not, i.e:

$\texttt{salary}^{bf}_{employee,integer}\texttt{(E,SAL)}$

Let E# be the employee number that corresponds to E. Execute the SQL query 'select salary from employee where enr=E#'. Let (E,s) be a result binding to (E,SAL), where <s> is the result tuple of the query.

4) Both the employee and the salary are known, i.e:

$\texttt{salary}^{bb}_{employee,integer}\texttt{(E,SAL)}$

Let E# be the employee number of E. Execute the SQL query 'select salary from employee where enr=E#'. If s=SAL then let (E,SAL) be a result binding to (E,SAL), where <s> is the result tuple of the query.

(End of description.)

This strategy would allow the salary function to be used in the backward direction, as in the following query:

```
select e for each employee e where salary(e)=15000
```

This would result in the execution of the following TBR predicate:

$\texttt{salary}^{fb}_{employee,integer}\texttt{(E,15000)}$

Which would result in a single SQL query being sent to the relational database:

```
select enr from employee where salary=15000
```

However, this strategy still does not allow an effective query optimization.

45. See footnote 44.

For example, consider the following AMOSQL query:

```
select name(e)
for each employee e
where salary(e)=15000
```

This would be translated to the following TR rule:

$$\begin{aligned}\mathtt{temp}_{charstring}(\mathtt{N}) \;\; \mathtt{<-} \;\; & \mathtt{name}_{employee,charstring}(\mathtt{E,N}) \;\&\\ & \mathtt{salary}_{employee,integer}(\mathtt{E,15000})\end{aligned}$$

Which would be translated to the following optimized TBR rule:

$$\begin{aligned}\mathtt{temp}^{f}_{charstring}(\mathtt{N}) \;\; \mathtt{<-} \;\; & \mathtt{salary}^{fb}_{employee,integer}(\mathtt{E,15000}) \;\&\\ & \mathtt{name}^{bf}_{employee,charstring}(\mathtt{E,N})\end{aligned}$$

The first predicate would result in the following SQL query being executed:

```
select enr from employee where salary=15000
```

For each of the results (say X) of this query, the following SQL query would be executed:

```
select name from employee where enr=X
```

It is obvious that an optimal translation should result in a single query against the relational database:

```
select name from employee where salary=15000
```

The problem is that access to the relational database is embedded in the code of the TBR predicates. This makes it impossible for the optimizer to generate optimal execution plans in the general case.

13.1.3 r-functions and r-predicates

For the optimizer to be able to reason about access to the relational database, it is essential that all relational database access is represented explicitly in query plans. We use a special kind of functions, *r-functions*, and a special kind of ObjectLog predicates, *r-predicates*, for this.

All functions in the object view which require access to the relational database are defined as derived functions which contain one or more calls to r-functions. The ObjectLog rules which correspond to these derived functions will contain one or more r-predicates.

For example, the function *salary(employee)->integer* discussed above is defined as follows:[46]

```
create function salary(employee e)->integer sal as
  select sal
  where oidmap(e)=enr
    and r_employee()=<enr,_,sal,_>
```

46. The oidmap function maps between employee objects and employee numbers. This will be discussed in section 13.2.

The function r_employee is an example of an r-function. Logically, an r-function returns all the tuples of a specific relation in the relational database.[47] For example, the r_employee() function call returns all the tuples of the employee relation.

The salary function will be compiled to the following TR rule:

```
salary_employee,integer(E,SAL)
                 <- oidmap_employee,integer(E,ENR) &
                    r_employee(ENR,_,SAL,_)
```

This can be read as 'SAL is the salary of E if ENR is the employee number of E and there is a tuple in the employee relation in the relational database where enr=ENR and salary=SAL'.

The predicate *r_employee(ENR,_,SAL,_)* is an example of an r-predicate. r-predicates are not executable as they stand. They are used as an intermediate form which the optimizer manipulates to decide the optimal combination of queries that are to be sent to the relational database.

As an example of query processing using this technique, consider again the query:

```
select name(e)
for each employee e
where salary(e)=15000
```

The query plan for this query will be:[48]

```
temp_charstring(N)  <- r_employee(ENR,N,_,_) &
                       r_employee(ENR,_,15000,_)
```

Since ENR is the key of the employee relation, the optimizer will unify the two r_employee predicates into a single one (this kind of compile-time unification is discussed in section 13.4.1):

```
temp_charstring(N)  <- r_employee(ENR,N,15000,_)
```

The final step of query optimization is to translate r-predicates into executable predicates that access the relational database (this is discussed in section 13.4.3). In this case, the final optimized ObjectLog program will be:

```
temp^f_charstring(N)  <- sql_exec^bf_charstring,charstring("select
                          name from employee where
                          salary=15000", N)
```

Which will result in a single SQL query being sent to the relational database.

---

47. In reality, they are only used as an intermediate form. There are no implementations of the r-functions.
The variable '_' is a special kind of variable which can be read as "don't care".
48. The query plan would also contain oidmap predicates (which map between employee numbers and OIDs). These predicates would be removed by the optimizer and are not included here. This is discussed in section 13.2 and section 13.4.2.

13.2 Object identity

A major difference between relational and object-oriented databases is that relational databases are *value based*, whereas object-oriented databases are *identity based*. The reason for calling object-oriented databases 'identity based' is that objects have an existence independent of the values of their attributes. Each object is uniquely identified by an object identifier (OID) which can always be used to refer to it. In contrast, if two tuples in a relational database have identical values for all attributes, the tuples are considered identical. This is usually handled by having a set of attributes (the primary key) whose values are always different for different real world objects.

In an object view of a relational database, there will be a correspondence between primary key values in the relational database and OIDs in the view. The Translator must generate OIDs which correspond to the different primary key values. It must also guarantee that a primary key value is mapped to the same OID each time it is accessed. This is a general problem for object views [1]. Suppose for example that an application issues a query which returns an OID (let us call this :obj). The application disconnects from the Translator but maintains the reference to :obj. The next time the application connects to the Translator, it issues a query which retrieves some property of :obj. Now, the Translator must map :obj to the same primary key value as when it was retrieved.[49]

For ease of presentation, we will only consider primary keys consisting of a single attribute. This can easily be extended to primary keys consisting of multiple attributes.

Two different approaches to OID management in object views can be distinguished; *algorithmic generation* of OIDs, and the use of OID *mapping tables*.

Algorithmic generation of OIDs means that the OID is somehow calculated based on the value of the primary key. The main benefit of this approach is that OIDs can be generated again and again, and one can still be sure that the same OID is generated each time a specific primary key value is accessed. A natural way to calculate the OID is to use a hash table. The problem with this kind of algorithmic generation is that collisions will occur - different values will sometimes generate identical OIDs. One way to guarantee that different values always generate different OIDs is to represent OIDs by a concatenation of the relation name and the primary key value.

---

49. Of course, it is impossible to guarantee that the primary key value is still in the relational database; some other application may have removed it. To guarantee this, the Translator would have to lock that tuple in the relational database for the entire lifetime of the application. This is not realistic. This, however, is a general problem in multi-user database systems.
Deletion semantics are discussed in section 13.2.4.

This is proposed as a plausible implementation in the Pegasus project [3]. The problem with this approach is that 'normal' OIDs and OIDs for mapped objects now have different representations.

In the mapping tables approach, there is no mathematical correspondence between the OID and the primary key value. There is no way to derive the primary key value from the OID, or vice versa. OIDs for mapped objects are generated dynamically the first time they are needed and are thereafter maintained by the Translator. The mapping between OIDs and primary key values is stored in internal tables in the Translator. We will refer to these tables as *oidmap tables*.

We use the mapping tables approach in AMOS and the rest of this section describes the principles behind the implementation of this.

Section 13.2.1 gives an introduction to OID management in AMOS Translators.

The mapping between OIDs and primary key values is modelled with a special kind of functions called *oidmap functions*. The corresponding ObjectLog predicates are called *oidmap predicates*. Section 13.2.2 describes what these functions and predicates should do. It also discusses the effects of implementing the oidmap functions directly as foreign functions. This would mean that calls to the relational database are embedded within the code of the oidmap predicates, which would make it impossible for the optimizer to generate optimal execution plans.

To avoid this, we define the oidmap functions as derived functions. The relational database access is handled by r-functions and the 'pure' OID mapping functionality is handled by another kind of functions which are called *oidmap1 functions*. The ObjectLog predicates corresponding to these functions are called *oidmap1 predicates*. These functions and predicates are described in section 13.2.3.

Finally, section 13.2.4 discusses what should happen when primary key values for which OIDs have been generated are deleted from the relational database.

13.2.1 oidmap tables

This section gives an introduction to how oidmap tables are used to manage the mapping between OIDs in AMOS Translators and the corresponding primary key values in the relational database.[50] Our examples will be taken from the company database.

Recall that we require the relational database schema to be in SSNF. This means that for each mapped type in the object view there is a relation in the

50. Recall that a Translator contains a complete AMOS database. This makes the oidmap tables persistent.

employee.oidmap

oid	enr

(a)

employee.oidmap

oid	enr
:e1	314

(b)

employee.oidmap

oid	enr
:e1	314
:e2	159
:e3	265
:e4	358

(c)

Figure 29: Content of the oidmap table for the type employee at different points in time. (a) No queries have retrieved any employees. (b) A query has retrieved the employee with employee number 314. (c) All employees have been retrieved.

relational database such that each tuple in the relation corresponds to an instance of the mapped type.

Suppose that T is a most general mapped type and that R is the relation that it is mapped to. Since the RDB schema is in SSNF, R contains the primary key values for *all* the instances of T, including the values that also correspond to instances of subtypes to T. This means that it suffices to have one oidmap table for each most general mapped type. No oidmap tables are needed for mapped types that are not a most general mapped type.

Since there is only one most general mapped type (employee) in the company example, there will be only one oidmap table (employee.oidmap). This table is shown in figure 29a. Before the object view has been used, the table is empty.

Suppose that the following query is the first one that is sent to the Translator:

```
select e for each employee e where salary(e)=20000
```

To answer this, the Translator will send the query

```
select enr from employee where salary=20000
```

to the relational database. The result of this query will be a relation consisting of a single value, the employee number 314. The Translator should now return the employee object which corresponds to the primary key value 314. Since the oidmap table is empty, a new object (:e1) has to be created. The fact that the object :e1 corresponds to the primary key value 314 is stored in the oidmap table (see figure 29b) and :e1 is returned as the answer to the initial query.

Now suppose that the following query is sent to the Translator:

```
select e for each employee e
```

The following query will be sent to the relational database:

```
select enr from employee
```

The result of this query is the relation {<314>,<159>,<265>,<358>}. The Translator should now return the employee objects which correspond to these primary key values. It first searches the oidmap table for the value 314 and finds that it corresponds to the object :e1. Hence, no new object has to be created for this primary key value. It continues in the same way with the other values (159, 265, and 358). None of these values are stored in the oidmap table which means that three new objects (:e2, :e3, and :e4) have to be created. The mapping between these new objects and their respective primary key values is stored in the oidmap table (see figure 29c). The four objects (:e1, :e2, :e3, and :e4) are returned as the answer to the initial query.

Now consider the following query to the Translator:

```
select s for each secretary s
```

This will result in the following query to the relational database:

```
select enr from secretary
```

The result of this query will be the relation {<358>}. Since secretary is a subtype to employee, which is a most general mapped type, the Translator searches the employee.oidmap table for the value 358 and finds that it corresponds to the object :e4. This object is returned as the answer to the initial query.

Finally, suppose that the following query is sent to the Translator:

```
select name(:e4)
```

The Translator will see that the object is a mapped object that belongs in the type tree having employee as its root.[51] It will therefore search the employee.oidmap table for the object :e4. The corresponding value is 358 which means that the following query will be sent to the relational database:

```
select name from employee where enr=358
```

13.2.2 oidmap functions and predicates

The mapping between OIDs and primary key values is implemented with the *oidmap function*. The oidmap function is overloaded and there is one resolvent function for each mapped type. An oidmap function takes a mapped object as argument and returns the primary key value that the object is mapped to:

```
create function oidmap(mt obj)->lt val as ...
```

51. The instance-of relationship for mapped objects depends on the state of the relational database. The only thing that is known to the Translator is what subtree of the type graph the object belongs in. To indicate this, the most general mapped type for the object is stored together with it. This is discussed in section 13.3.

For example:

```
create function oidmap(employee e)->integer enr as ...
```

All functions in the object view which have a mapped type in their signature are implemented as derived functions which make a call to the oidmap function.

For example, the function *salary(employee)->integer* is defined as the following derived function:

```
create function salary(employee e)->integer sal as
  select sal
  where oidmap(e)=enr
    and r_employee()=<enr,_,sal,_>
```

Which will be translated to the following TR rule:

$$\texttt{salary}_{employee,integer}\texttt{(E,SAL)} \leftarrow \texttt{oidmap}_{employee,integer}\texttt{(E,ENR)} \ \& \ \texttt{r_employee(ENR,_,SAL,_)}$$

This can be read as 'SAL is the salary of E if ENR is the employee number of E and there is a tuple in the relational database where enr=ENR and salary=SAL'.

The oidmap functions are defined as derived functions, for reasons that will be explained in this section. To motivate this, we will start by describing what would have happened *if* the oidmap functions had been implemented directly as foreign functions (rather than derived functions).

Semantics of oidmap predicates

The following is an informal description of what the semantics of the different TBR predicates for the oidmap functions would have been if they had been implemented as foreign functions. *mt* is the mapped type on which the oidmap function is defined. *rel* is the relation that mt is mapped to. *lt* is the literal type corresponding to the domain of the primary key attribute of rel.

1) Neither the mapped object nor the primary key value are known, i.e.:

$\texttt{oidmap}^{ff}_{mt,lt}\texttt{(OBJ,VAL)}$

Get all primary key values from rel. For each of these values (v), check if there is a tuple <o, v> in the oidmap table for mgmt(mt). If there is, let (o,v) be a result binding to (OBJ,VAL). If there is not, create a new object (o_new) of the type mgmt(mt)[52], let (o_new,v) be a result binding to (OBJ,VAL), and add the tuple <o_new, v> to the oidmap table for

52. Since the instance-of relationship for mapped objects depends on the state of the relational database, only the most general mapped type for a mapped object is stored together with it (see footnote 51). To create a new object of the type mgmt(mt) means 'to create a new mapped object and declare that it belongs in the type tree having mgmt(mt) as its root'.

mgmt(mt).

2) The primary key value is known but the mapped object is not, i.e.:

$\texttt{oidmap}^{fb}_{mt,lt}\texttt{(OBJ,VAL)}$

Check that VAL is still a primary key value in rel.[53] If it is not, stop. Check if there is a tuple <o, VAL> in the oidmap table for mgmt(mt). If there is, let (o,VAL) be a result binding to (OBJ,VAL). If there is not, create a new object (o_new) of the type mgmt(mt), let (o_new,VAL) be a result binding to (OBJ,VAL), and add the tuple <o_new, VAL> to the oidmap table for mgmt(mt).

3) The mapped object is known, but the primary key value is not, i.e.:

$\texttt{oidmap}^{bf}_{mt,lt}\texttt{(OBJ,VAL)}$

Get the tuple <OBJ, v> in the oidmap table for mgmt(mt). Check that v is still a primary key value in rel. If it is not, stop. If it is, let (OBJ,v) be a result binding to (OBJ,VAL).

4) Both the mapped object and the primary key value are known, i.e.:

$\texttt{oidmap}^{bb}_{mt,lt}\texttt{(OBJ,VAL)}$

Check that there is a tuple <OBJ, VAL> in the oidmap table for mgmt(mt). If there is not, stop. Check that VAL is still a primary key value in rel. If it is, let (OBJ,VAL) be a result binding to (OBJ,VAL).

(End of description.)

Note that resolvent predicates 1 and 2 sometimes have side effects (new objects are created and an oidmap table is updated).

Break out relational database access from oidmap predicates

All the TBR predicates for oidmap require access to the relational database.[54] If the TBR predicates were directly implemented according to the semantics above, it would mean that relational database access would be embedded in the code of the predicates. As discussed in section 13.1, it is essential that all relational database access is represented explicitly in query plans. Otherwise, some queries will be processed in a non-optimal way.

To illustrate this, we will take the following query as an example:

```
select salary(:e1)
```

This would be translated to the following TBR rule:

$$\texttt{temp}^{f}_{integer}\texttt{(SAL)} \;\texttt{<-}\; \texttt{oidmap}^{bf}_{employee,integer}\texttt{(:e1,ENR)}\;\texttt{\&}\; \texttt{r_employee(ENR,_,SAL,_)}$$

53. This check could be skipped (in this resolvent predicate as well as in the following), which would lead to a different semantics for oidmap predicates. In that case, a mapping (o,v) in the oidmap table would be returned even if v had been deleted from rel.

54. If the alternative semantics discussed in footnote 53 had been chosen, only the first resolvent predicate (’ff’) would require access to the relational database.

Execution of the oidmap predicate would involve the execution of the following query (to check that 314, the enr of :e1, is still in the relational database):[55]

```
select T from employee where enr=314
```

The r_employee predicate would lead to the execution of the following query (to retrieve the salary for :e1):

```
select salary from employee where enr=314
```

Clearly, the first query is unnecessary, since the second query does the job of the first query (and more). However, the optimizer has no possibilities to discover this since the first query is embedded within the code of the oidmap predicate.

Note that all four resolvent predicates above consist of two parts. One part deals with the state of the relational database. It sends queries to get all primary key values for a relation or to check whether a certain value exists in the database. The other part deals with the mapping between OIDs and values. It searches and updates the oidmap tables.

This makes it possible to break out the relational database access and have it explicitly represented in the query plan. The oidmap functions are defined as derived functions. The relational database access is handled by r-functions and access to the oidmap tables is handled by the function oidmap1:

```
create function oidmap(mt o)->lt val as
  select val
  where val=oidmap1(o)
    and r_rel()=<..., val, ...>;
```

For example:

```
create function oidmap(employee e)->integer enr as
  select enr
  where enr=oidmap1(e)
    and r_employee()=<enr,_,_,_>;

create function oidmap(salesman s)->integer enr as
  select enr
  where enr=oidmap1(s)
    and r_salesman()=<enr,_,_>;
```

The oidmap1 function is the subject of the next section.

13.2.3 oidmap1 functions and predicates

Just like the oidmap functions, an oidmap1 function takes a mapped object

55. Since the query is used only to test whether a certain value is in the database, it does not matter what it returns, the important thing is that something is returned. Hence 'select T ...'.

as argument and returns the primary key value that the object is mapped to. However, unlike oidmap functions, oidmap1 functions do not involve access to the relational database. They are only concerned with access to the oidmap table. The oidmap1 function is overloaded and there is one resolvent function for each most general mapped type:

```
create function oidmap1(mgmt obj)->lt val as ...
```

For example:

```
create function oidmap1(employee e)->integer enr as ...
```

There are no resolvents of the oidmap1 function for subtypes to most general mapped types. These types inherit the oidmap1 function.

The oidmap functions are compiled to the following kind of TR rules:

$$\texttt{oidmap}_{mt,lt}\texttt{(O,VAL)} \leftarrow \texttt{oidmap1}_{mgmt(mt),lt}\texttt{(O,VAL)} \;\&\; \texttt{r_rel(...,VAL,...)}$$

For example:

$$\texttt{oidmap}_{employee,integer}\texttt{(E,ENR)} \leftarrow \texttt{oidmap1}_{employee,integer}\texttt{(E,ENR)} \;\&\; \texttt{r_employee(ENR,_,_,_)}$$

$$\texttt{oidmap}_{salesman,integer}\texttt{(S,ENR)} \leftarrow \texttt{oidmap1}_{employee,integer}\texttt{(S,ENR)} \;\&\; \texttt{r_salesman(ENR,_,_)}$$

When the oidmap function is used together with other functions, the r-predicates will most often be optimized away (unified with other r-predicates). This is discussed in section 13.4.1.

Semantics of oidmap1 predicates

The following is an informal description of the semantics of the different TBR predicates for the oidmap1 functions. *mt* is the (most general) mapped type on which the oidmap1 function is defined. *rel* is the relation that mt is mapped to. *lt* is the literal type corresponding to the domain of the primary key attribute of rel.

1) Neither the mapped object nor the primary key value is known, i.e.:

$$\texttt{oidmap1}^{ff}_{mt,lt}\texttt{(OBJ,VAL)}$$

These resolvent predicates are not needed. Since oidmap1 predicates are always accompanied by r-predicates (see above) the optimizer can always chose to execute the oidmap1 predicate after the r-predicate. This means that the second argument to oidmap1 is bound and that the resolvent predicate

$$\texttt{oidmap1}^{fb}_{mt,lt}\texttt{(OBJ,VAL)}$$

is the one that is executed.

2) The primary key value is known but the mapped object is not, i.e.:

$\texttt{oidmap1}^{fb}_{mt,lt}\texttt{(OBJ,VAL)}$

Check if there is a tuple <o, VAL> in the oidmap table for mt. If there is, let (o,VAL) be a result binding to (OBJ,VAL). If there is not, create a new object (o_new) of the type mt, let (o_new,VAL) be a result binding to (OBJ,VAL), and add the tuple <o_new, VAL> to the oidmap table for mt.

3) The mapped object is known, but the primary key value is not, i.e.:

$\texttt{oidmap1}^{bf}_{mt,lt}\texttt{(OBJ,VAL)}$

Get the tuple <OBJ, v> in the oidmap table for mt. Let (OBJ,v) be a result binding to (OBJ,VAL).

4) Both the mapped object and the primary key value are known, i.e.:[56]

$\texttt{oidmap1}^{bb}_{mt,lt}\texttt{(OBJ,VAL)}$

Check that there is a tuple <OBJ, VAL> in the oidmap table for mt. If there is, let (OBJ,VAL) be a result binding to (OBJ,VAL).

(End of description.)

Again, consider the example query from section 13.2.2:

```
select salary(:e1)
```

The query will be translated to the following TR rule:

$$\texttt{temp}_{integer}\texttt{(SAL)} \;\texttt{<-}\; \texttt{oidmap}_{employee,integer}\texttt{(:e1,ENR)}\ \texttt{\&}\ \texttt{r_employee(ENR,_,SAL,_)}$$

Since oidmap is a derived function, this will be further translated to the following (fully expanded) TR rule:

$$\texttt{temp}_{integer}\texttt{(SAL)} \;\texttt{<-}\; \texttt{oidmap1}_{employee,integer}\texttt{(:e1,ENR)}\ \texttt{\&}\ \texttt{r_employee(ENR,_,_,_)}\ \texttt{\&}\ \texttt{r_employee(ENR,_,SAL,_)}$$

The two r_employee predicates can be unified into a single one (this is discussed in section 13.4.1). Actually, this is what usually happens to the r-predicates that accompany oidmap1 predicates - they are removed by the optimizer. This gives the following TR rule:

$$\texttt{temp}_{integer}\texttt{(SAL)} \;\texttt{<-}\; \texttt{oidmap1}_{employee,integer}\texttt{(:e1,ENR)}\ \texttt{\&}\ \texttt{r_employee(ENR,_,SAL,_)}$$

Which will be translated to the following TBR rule:

$$\texttt{temp}^{f}_{integer}\texttt{(SAL)} \;\texttt{<-}\; \texttt{oidmap1}^{bf}_{employee,integer}\texttt{(:e1,ENR)}\ \texttt{\&}\ \texttt{r_employee(ENR,_,SAL,_)}$$

Execution of the oidmap1 predicate would not involve the execution of any

56. Note that this TBR predicate is not necessary, since it is covered [47] by the third TBR predicate ('bf'). Instead of executing the 'bb' predicate, the 'bf' predicate could be executed followed by an equality test that checked the retrieved primary key value.

SQL queries. As before, the r_employee predicate would lead to the execution of the following query (to retrieve the salary for :e1):

```
select salary from employee where enr=314
```

Side effects of oidmap1 predicates

The resolvent predicates

$\texttt{oidmap1}^{fb}_{mt,lt}\texttt{(OBJ,VAL)}$

sometimes have side effects. This happens when no object corresponding to VAL is stored in the oidmap table for mt. In this case a new object (o_new) is created and the tuple <o_new, VAL> is added to the oidmap table.

This could sometimes cause the creation of unnecessary objects and that unnecessary tuples are added to the oidmap tables. For example, consider the following query:

```
select e for each employee e where enr(e)=777
```

One of the possible TBR alternatives is:

$\texttt{temp}^{f}_{employee}\texttt{(E)} \quad \texttt{<- oidmap1}^{fb}_{employee,integer}\texttt{(E,777) \&}$
$\qquad\qquad\qquad\quad \texttt{r_employee(777,_,_,_)}$

Suppose that this TBR alternative was chosen by the optimizer. No object corresponding to 777 would be stored in the employee.oidmap table. This means that a new object :e5 would be created and that the tuple <:e5, 777> would be added to the employee.oidmap table. After this, the r_employee predicate would cause the Translator to check if a tuple with enr=777 exists in the employee relation. Since no such tuple exists, the query would not return any employees. Still, the side effect of the oidmap1 predicate has resulted in the creation of an object and a new tuple in an oidmap table.

The side effects do not affect the semantics of the object view since oidmap1 predicates are always accompanied by r-predicates that check that values found in the oidmap table are present in the relational database. For example, consider a query like

```
select e for each employee e
```

This will be translated to the following TR rule:

$\texttt{temp}_{integer}\texttt{(E)} \quad \texttt{<- oidmap1}_{employee,integer}\texttt{(E,ENR) \&}$
$\qquad\qquad\qquad\quad \texttt{r_employee(ENR,_,_,_)}$

If the optimizer should choose to execute the oidmap1 predicate first, one branch of the top-down execution tree will have E bound to :e5 and ENR bound to 777. However, since the r_employee predicate will check that there is a tuple in the employee relation with enr=777 (which there is not), :e5 will not be one of the answers to the query. The only problem with these side effects is that some extra memory is used. Since queries which result in unwanted side effects are very rare and artificial, we consider this problem negligible.

13.2.4 Deletion semantics

This section discusses possible semantics for the object view when primary key values in the relational database for which OIDs have been generated are deleted. In the following discussion, :obj is a mapped object and MT is the mapped type which :obj is a direct instance of.

Let us first consider the case when the primary key value is deleted from a relation which is mapped to a type which is not a most general mapped type. This causes no problems - the only thing that happens is that the type membership for :obj changes. Instead of being a direct instance of MT, it will be a direct instance of the supertype of MT. For example, if the tuple where enr=314 is deleted from the salesman relation, the object :e1 will no longer be an instance of the type salesman, but it will still be an instance of the type employee.[57]

The problems start when (1) MT is a most general mapped type, and (2) some application has a handle to :obj. Suppose for example that some application has issued a query against the object view which returned the object :e1, that the application keeps a reference to this object, and that later, the tuple where enr=314 is deleted from the employee relation. The question is what should happen when the application issues a new query which involves the object :e1, for example:

```
select salary(:e1)
```

In the current prototype, a query like this will simply return an empty result. The query will be translated to the following TBR rule:

$$\mathtt{temp}^{f}_{integer}\mathtt{(SAL)} \;\; \mathtt{<\!-}\; \mathtt{oidmap1}^{bf}_{employee,integer}\mathtt{(:e1,ENR)}\;\&$$
$$\mathtt{r_employee(ENR,_,SAL,_)}$$

The oidmap1 predicate will bind the variable ENR to 314, since the mapping between :e1 and the primary key value 314 is still stored in the oidmap table. The r_employee predicate will therefore result in the following query against the relational database:

```
select salary where enr=314
```

Since there is no tuple where enr=314, the initial query will not return anything.

Different semantics are possible for a situation like this. Maybe the Translator should notify the application that the reference to :obj is obsolete, rather than just return an empty result for queries like the one above. The notification could either be performed the first time the application uses :obj in a query, or as soon as the primary key value is deleted. The latter case would require an active database mechanism [28] [58] or some kind of monitoring [54] of the relational database.

A related question is what should happen if the primary key value is added

57. Type membership of mapped objects is discussed in section 13.3.

to the relational database again. In the current prototype, the primary key value is mapped to the same OID and applications are unaffected by the fact the value was deleted for a period of time (unless they tried to use it while it was absent). Different semantics are possible for this too.

13.3 The instance-of relationship

Query plans in object-oriented databases often contain tests of the *instance-of relationship*, i.e. which objects are instances of which types. We will refer to these tests as *type membership tests*. They may, for example, be used to get the types that an object is an instance of, or to get all objects that are instances of a specific type.

In AMOS, the instance-of relationship is modelled by the *typesof function*. It takes an object as argument and returns the types that the object is an instance of (direct or by generalization):

```
typesof(object)->type
```

When queries are compiled to ObjectLog rules, calls to the typesof function are replaced with *typesof predicates*:

$\texttt{typesof}_{object,type}\texttt{(OBJ,TP)}$

There are four categories of queries which result in type membership tests in AMOS query plans:

1) Direct call

```
select typesof(:e1)
```

This is translated to the following TR rule:

$\texttt{temp}_{type}\texttt{(T)} \quad \texttt{<- typesof}_{object,type}\texttt{(:e1,T)}$

2) Get all instances of a type

```
select e for each employee e
```

This is translated to the following TR rule:

$\texttt{temp}_{employee}\texttt{(E)} \quad \texttt{<- typesof}_{object,type}\texttt{(E,:typeEmployee)}$

3) Inherited functions

```
select salary(s) for each secretary s
```

This is translated to the following TR rule:

$\texttt{temp}_{integer}\texttt{(SAL)} \quad \texttt{<- salary}_{employee,integer}\texttt{(S,SAL) \&}$
$\quad\quad \texttt{typesof}_{object,type}\texttt{(S,:typeSecretary)}$

4) Late binding

Suppose that the function *salary* is defined not only on the type employee, but also on the type secretary (*salary* is an overloaded function). Then the query

```
select salary(e) for each employee e
```

is translated to the following TR rule:

$\texttt{temp}_{integer}\texttt{(SAL)} \quad \texttt{<- typesof}_{object,type}\texttt{(E,:typeEmployee) \&}$
$\quad\quad \texttt{apply(:functionSalary,E,SAL)}$

At execution time, the *typesof* predicate will retrieve all employee objects. For each of these objects, the *apply* predicate will perform another type

membership test to decide what resolvent of the *salary* function should be chosen.

(End of enumeration.)

In an object-oriented database, the relationship between objects and types (which objects are instances of which types) can be stored directly in the database. This makes it possible to implement type membership tests very efficiently.

In an object view of a relational database, the relationship between objects and types is dependent on the state of the relational database. Type membership tests require access to the relational database. This makes these tests very expensive, and it essential that they are removed by the optimizer whenever possible.

A mapped object OBJ is an instance of a mapped type TP if: (a) OBJ is mapped to a primary key value that occurs in the relation that TP is mapped to, and (b) mgmt(TP) is equal to mgmt(OBJ). Condition (b) is necessary to handle cases where a primary key value occurs in two relations which are mapped to types with different most general mapped types. For example, an employee object which is mapped to the primary key value enr=314 is not an instance of the type department even if there is a department with the primary key value dnr=314.

Section 13.3.1 discusses the role of the typesof function in AMOS Translators.

13.3.1 The typesof function

The typesof function is implemented as a foreign function. As will be shown in this section, all TBR predicates for the function contain calls to the relational database. It should be clear by now that this causes problems for the optimizer - it will not have enough information to generate optimal execution strategies in all cases.[58] We will illustrate this and present a solution to part of the problem. However, first we will give a more detailed description of the typesof function.

Semantics of typesof predicates
The following is an informal description of the semantics of the different TBR predicates for the typesof function.

Note that the object (OBJ) may be either a mapped object or a regular object, and that the type (TP) may be a mapped type or a regular type. Since our focus is on object views of relational data, we only describe the cases of mapped objects and types. That is, when OBJ is known, it is bound to a mapped object, and when TP is known, it is bound to a mapped type. In general, when the objects/types are not mapped, a standard object-oriented

58. See section 13.1 and section 13.2.

database implementation of the typesof predicates can be used.[59]

1) Neither the mapped object nor the primary key value is known, i.e.:

$\texttt{typesof}^{ff}_{object,type}\texttt{(OBJ,TP)}$

This resolvent predicate is not needed. It is always possible to choose an ordering of the query plan predicates so that either the object or the type is known when the typesof-predicate is executed. A query like

```
select typesof(o) for each object o
```

is translated to the following TR rule:

$\texttt{temp}_{type}\texttt{(T)} \quad \texttt{<-}\ \texttt{typesof}_{object,type}\texttt{(O,:typeObject) \&}$
$\qquad\qquad\qquad \texttt{typesof}_{object,type}\texttt{(O,T)}$

2) The type is known but the object is not, i.e.:

$\texttt{typesof}^{fb}_{object,type}\texttt{(OBJ,TP)}$

Let rel be the relation that TP is mapped to. Get all primary key values from rel. For each of these values (v), check if there is a tuple <o, v> in the oidmap table for mgmt(TP). If there is, let (o,TP) be a result binding to (OBJ,TP). If there is not, create a new object (o_new) of the type mgmt(TP), let (o_new,TP) be a result binding to (OBJ,TP), and add the tuple <o_new, v> to the oidmap table for mgmt(TP).

3) The object is known but the type is not, i.e.:

$\texttt{typesof}^{bf}_{object,type}\texttt{(OBJ,TP)}$

Let t_0 be mgmt(OBJ). Get the tuple <OBJ, v> in the oidmap table for t_0. Let T be the subtree of the type tree that has t_0 as its root. Traverse T top-down and for each type t that is a node of T do the following: Let rel be the relation that t is mapped to. Check if v is a primary key value in rel. If it is, let (OBJ,t) be a result binding to (OBJ,TP). If it is not, do not traverse the subtree having t as its root any further.

4) Both the object and the type are known, i.e.:

$\texttt{typesof}^{bb}_{object,type}\texttt{(OBJ,TP)}$

Let t_0 be mgmt(OBJ). Check if TP is either equal to t_0 or a subtype to t_0. If it is not, stop. If it is, get the tuple <OBJ, v> in the oidmap table for t_0. Let rel be the relation that TP is mapped to. Check if v is a primary key value in rel. If it is, let (OBJ,TP) be a result binding to (OBJ,TP).

Break out relational database access from typesof predicates

All the TBR predicates for the typesof function require access to the relational database. As discussed in section 13.1 and section 13.2, this is a problem during query optimization. Some queries will be processed in a non-optimal way.

---

59. The only tricky case is when the type is a regular type which can have mapped subtypes. Consider for example the query 'select o for each object o'. The query should return *all* objects, both regular and mapped.

For example, consider the query:

```
select salary(s) for each secretary s
```

This would be translated to the following TBR rule:

$$\texttt{temp}^{f}_{integer}\texttt{(SAL)} \;\; \texttt{<- r_employee(ENR,_,SAL,_) \&}$$
$$\texttt{oidmap1}^{fb}_{employee,integer}\texttt{(S,ENR) \&}$$
$$\texttt{typesof}^{bb}_{object,type}\texttt{(S,:typeSecretary)}$$

The r_employee predicate would lead to the execution of the following SQL query (to get the employee number and salary for all employees):

```
select enr, salary from employee
```

The oidmap1 predicate would search the employee.oidmap table to get the employee objects (four) that correspond to the employee numbers. This would result in four executions of the typesof predicate (one execution for each employee object).

Let us consider execution of the typesof predicate when S is bound to the object :e1. First, the employee number of :e1 (314) would again be retrieved from the employee.oidmap table. After this, the following SQL query would be executed (to check if 314, the enr of :e1, is in the secretary table):

```
select T from secretary where enr=314
```

Since this query would not return anything, the salary of :e1 would not be an answer to the initial query. The same procedure is repeated for the other employee objects (and the salary of :e4 is the only thing that is returned).

Clearly, this is a non-optimal execution strategy. An optimal execution strategy would result in a single SQL query:

```
select salary
from employee, secretary
where employee.enr=secretary.enr
```

The problem is that relational database access is embedded in the code of the typesof TBR predicates. Fortunately, there is a way to overcome some of these problems.

Recall that a mapped object OBJ is an instance of a mapped type TP if OBJ is mapped to a primary key value VAL, VAL occurs in the relation that TP is mapped to, and mgmt(TP) is equal to mgmt(OBJ).

This can be expressed as the following rule, where rel is the relation that TP is mapped to:[60]

60. The condition that mgmt(TP) is equal to mgmt(OBJ) is guaranteed by the oidmap1 predicate. If mgmt(TP)≠mgmt(OBJ) then OBJ can never occur in the oidmap table for mgmt(TP).

typesof$_{object,type}$(OBJ,TP)
  <- oidmap1$_{mgmt(TP),lt}$(OBJ,VAL) &
    r_rel(...,VAL,...)

This rule can be used to replace typesof predicates at compile-time in the cases where TP is a constant.[61]

For example:

typesof$_{object,type}$(E,:typeSecretary)
  <- oidmap1$_{employee,integer}$(E,ENR) &
    r_secretary(ENR,_)

Again consider the query

```
select salary(s) for each secretary s
```

The TR rule for this query is:

temp$_{integer}$(SAL) <- r_employee(ENR,_,SAL,_) &
    oidmap1$_{employee,integer}$(S,ENR) &
    typesof$_{object,type}$(S,:typeSecretary)

Since the second argument to the typesof-predicate is a constant, the typesof substitution rule above can be applied to the rule, which becomes:

temp$_{integer}$(SAL) <- r_employee(ENR,_,SAL,_) &
    oidmap1$_{employee,integer}$(S,ENR) &
    oidmap1$_{employee,integer}$(S,ENR2) &
    r_secretary(ENR2,_)

Compile-time unification (see section 13.4.1) and removal of unnecessary oidmap1 predicates (see section 13.4.2) gives:

temp$_{integer}$(SAL) <- r_employee(ENR,_,SAL,_) &
    r_secretary(ENR,_)

Which will give the following executable TBR rule (see section 13.4.3):

temp$^{f}_{integer}$(SAL) <- sql_exec$^{bf}_{charstring,integer}$("select salary
      from employee, secretary where
      employee.enr=secretary.enr", SAL)

Remaining problems with type membership tests

The only occasion when typesof predicates can be substituted away is when the type is known at compile-time. This means that some queries are still processed in a non-optimal way. Consider for example the following query:

```
select salary(:e1), typesof(:e1)
```

This is translated to the following TR rule:

61. This is performed by the typesof Expander (see figure 30 on page 99).

```
temp_integer,type(SAL,TP)
                <- oidmap1_employee,integer(:e1,ENR) &
                   r_employee(ENR,_,SAL,_) &
                   typesof_object,type(:e1,TP)
```

The enr of *:e1* is 314. The *r_employee* predicate will therefore result in the execution of the following SQL query:

```
(a) select salary from employee where enr=314
```

The *typesof* predicate is used to get all types that *:e1* is an instance of, i.e. the types corresponding to those relations in which 314 is a primary key value. To do this it will execute the following SQL queries:

```
(b) select T from employee where enr=314
(c) select T from secretary where enr=314
(d) select T from salesman where enr=314
```

Clearly, query (b) is unnecessary, since the job of it (and more) has already been done by query (a).

Our query processing technique is not powerful enough to produce optimal query plans in all cases. Note however that it *can* handle queries of type (2) and (3) in the enumeration on page 92. Our experience is that these types of queries are, by far, the most common among queries that generate type membership tests.

13.4 Query optimization

As discussed in section 12.3, the optimizer reorders the subgoals of TR rules to the most efficient execution strategy. The estimation of which reordering is optimal is based on the cost model described in [47].

Prior to query optimization, two techniques can be used to simplify the TR rule. The two techniques, compile-time unification of predicates and removal of unnecessary oidmap1 predicates, are discussed in section 13.4.1 and section 13.4.2, respectively.

TR rules contain r-predicates, which are not executable as they stand. After query optimization, they are replaced by executable sql_exec predicates which make the calls to the relational database. Sometimes, several r-predicates should be combined and replaced by a single sql_exec predicate. Section 13.4.3 concerns the substitution of r-predicates.

Finally, section 13.4.4 gives a longer example of query processing in object views of relational data.

Figure 30 gives an overview of the different phases of query processing in Translators (compare this to query processing in normal AMOS databases which is illustrated in figure 26 on page 69).

13.4.1 Compile-time unification

It is sometimes possible to unify predicates in a query plan and replace them with a single predicate. This is the case when two predicates have the same predicate symbol, have the same constants/variables in the key attributes, and there are no conflicts between constants in the non-key attributes.

For example, consider the following TR rule:

$$temp_{integer}(SAL_1) \ \text{<-}\ salary_{employee,integer}(E,SAL_1)\ \&\ salary_{employee,integer}(E,SAL_2)\ \&\ foo_{integer}(SAL_2)$$

The first argument of salary predicates is a key since employees can only have one salary. Since the two salary predicates have the same variable (E) as their first argument, they can be unified and replaced with a single predicate. The resulting substitution (SAL_2 should be replaced by SAL_1) is applied to the rest of the predicates. This gives the following TR rule:

$$temp_{integer}(SAL_1) \ \text{<-}\ salary_{employee,integer}(E,SAL_1)\ \&\ foo_{integer}(SAL_1)$$

This type of compile-time unification is important for queries against an object view of a relational database. Query plans will often contain multiple r-predicates and multiple oidmap1 predicates which can be unified. For example, consider the following query:

```
select name(:e1), salary(:e1)
```

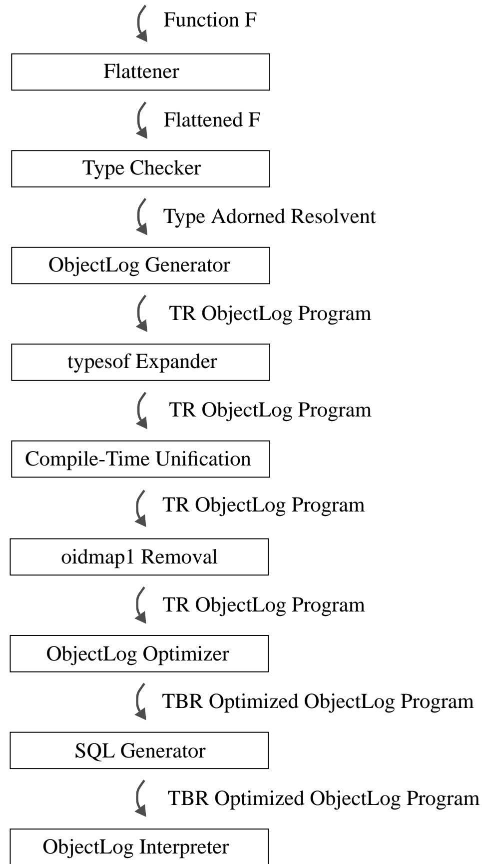


Figure 30: Query processing steps in Translators.
typesof Expander see section 13.3. *Compile-Time Unification* see section 13.4.1. *oidmap1 Removal* see section 13.4.2. *ObjectLog Optimizer* see section 12.3 and section 13.4.3. *SQL Generator* see section 13.4.3.

The ObjectLog generator will first transform this to the following (not final) TR rule:

$$\texttt{temp}_{\texttt{charstring,integer}}\texttt{(N,SAL)} \leftarrow \texttt{name}_{\texttt{employee,charstring}}\texttt{(:e1,N)}\ \&\ \texttt{salary}_{\texttt{employee,integer}}\texttt{(:e1,SAL)}$$

Since name and salary are derived functions, this will be expanded to the following TR rule:

```
temp_charstring,integer(N,SAL)
                  <- oidmap1_employee,integer(:e1,ENR_1) &
                     r_employee(ENR_1,N,_,_) &
                     oidmap1_employee,integer(:e1,ENR_2) &
                     r_employee(ENR_2,_,SAL,_)
```

Since both oidmap1 predicates have the same constant (:e1) as their first argument, which is a key, they can be unified. Each occurrence of the variable ENR_2 is replaced with ENR_1. This gives the following TR rule:

```
temp_charstring,integer(N,SAL)
                  <- oidmap1_employee,integer(:e1,ENR_1) &
                     r_employee(ENR_1,N,_,_) &
                     r_employee(ENR_1,_,SAL,_)
```

Now, the two r_employee predicates have the same variable (ENR1) as their first argument, which is a key. This means that they too can be unified. The final TR rule therefore looks like this:

```
temp_charstring,integer(N,SAL)
                  <- oidmap1_employee,integer(:e1,ENR_1) &
                     r_employee(ENR_1,N,SAL,_)
```

13.4.2 Removal of oidmap1 predicates

It is sometimes possible to remove oidmap1 predicates from the query plan without affecting the semantics of the query. We define the rule for this as follows:

If: The first argument of an oidmap1 predicate is a variable which does not occur in any other predicate of the query plan (neither in the head nor in any subgoal)

Then: The oidmap1 predicate can be removed from the query plan.

Consider for example the following query:

```
select name(e), salary(e) for each employee e
```

This will be translated to the following TR rule:

```
temp_charstring,integer(N,SAL)
                  <- oidmap1_employee,integer(E,ENR_1) &
                     r_employee(ENR_1,N,_,_) &
                     oidmap1_employee,integer(E,ENR_2) &
                     r_employee(ENR_2,_,SAL,_)
```

Compile-time unification (see section 13.4.1) gives:

```
temp_charstring,integer(N,SAL)
                <- oidmap1_employee,integer(E,ENR_1) &
                   r_employee(ENR_1,N,SAL,_)
```

Since the variable E of the oidmap1 predicate does not occur in any other predicate, the oidmap1 predicate can be removed. This gives the following, final, TR rule:

```
temp_charstring,integer(N,SAL)
                <- r_employee(ENR_1,N,SAL,_)
```

Of course, this kind of removal of predicates can not be applied to any type of predicates. Consider for example the following query, which retrieves all employees that have a hobby:

```
select e
for each employee e, charstring h
where hobby(e)=h
```

The final TR rule for this query will be:

```
temp_employee(E)    <- r_emp_hobbies(ENR,H) &
                       oidmap1_employee,integer(E,ENR)
```

The variable H of the r_emp_hobbies predicate does not occur in any other predicate. This does not mean that the r_emp_hobbies predicate can be removed. It serves as a boolean test that the employee *does* have a hobby.

Before we motivate why the removal *can* be applied to oidmap1 predicates, observe that the only case we need to discuss is when the first argument of oidmap1 (the oid variable) is unbound and the second argument (the primary key value) is bound, i.e:

$\texttt{oidmap1}^{fb}_{mt,lt}\texttt{(OBJ,VAL)}$

If the oid variable is bound, it is either a constant, or it has been bound by some other predicate. In any case, it violates the precondition of the removal rule; that the first argument should be a variable that does not occur in any other predicate.

Also recall from section 13.2.3 that oidmap1 predicates where both arguments are unbound never occur.

The reason why oidmap1 predicates can be removed is that they are never used as boolean tests in the way the r_emp_hobbies predicate was used in the example above. The only purpose of an oidmap1 predicate is to map between oid's and primary key values. If some value does not have a corresponding oid, a new oid is created as a side effect of the oidmap1 predicate.

Another way to look at the removal rule is as follows. Consider the following reordering of the subgoals of the first TR rule of this section:

```
temp_charstring,integer(N,SAL)
                <- r_employee(ENR_1,N,_,_) &
                   oidmap1_employee,integer(E,ENR_1) &
                   oidmap1_employee,integer(E,ENR_2) &
                   r_employee(ENR_2,_,SAL,_)
```

Suppose that the query was not optimized any further, and that the predicates were executed in this order. The first oidmap1 predicate would then take a primary key value and retrieve the corresponding oid. The second oidmap1 predicate would then take this oid and, again, retrieve the corresponding primary key value. Obviously, these two operations are not necessary. Using this viewpoint, the removal rule can be described as follows, using a functional notation:[62]

```
oidmap1(oidmap1^-1(OBJ)) <=> OBJ
```

If we think of the OID of a mapped object as an abstraction of the corresponding primary key value, we can see that the removal rule is actually a special case of the following, more general, rule:

```
deabstract(abstract(OBJ)) <=> OBJ
```

13.4.3 Substitution of r-predicates

TR rules contain r-predicates which represent access to the relational database. These predicates are not executable as they stand. The optimizer substitutes them into executable *sql_exec predicates*.

The first argument of an sql_exec predicate is the SQL query that is to be sent to the relational database. The query may contain variables on the form $!v_X$, where X=1, 2, 3, etc. After this follows zero, one or more arguments which are bound. These are substituted into the SQL query at execution time. The first of these arguments replaces the variable $!v_1$, the second replaces $!v_2$, etc. Finally follows zero, one or more unbound arguments. Execution of the SQL query results in these arguments being bound.

For example, consider a query plan that contains the following r-predicate:

```
r_employee(ENR,NAME,SAL,_)
```

If the optimizer chooses an execution order where NAME is bound whereas ENR and SAL are unbound, this will result in the following sql_exec predicate:

```
sql_exec("select enr,salary from employee where name=!v1",
  NAME,ENR,SAL)
```

Let us assume that NAME is bound to 'bertil' at execution time. Then the

62. Actually, if algebraic expressions had been used as the internal representation of query plans, there would have been a transformation rule of this kind that corresponded to our removal rule.
oidmap1^{-1} is the inverse of the *oidmap1* function.

following query will be sent to the relational database:

```
select enr,salary from employee where name='bertil'
```

The query returns the tuple <159, 15000>. This results in ENR being bound to 159 and SAL being bound to 15000.

A naive substitution algorithm would be to simply replace each r-predicate with an sql_exec predicate, in the way it was done above. This naive algorithm is actually the one that is implemented in the current prototype. We will present the details of this algorithm in the next section. In the general case, the algorithm is not capable of creating optimal query plans. We will discuss its shortcomings and possible improvements in the sections following the next one.

Naive substitution

Each r-predicate

```
r_rel(W1,...,Wn)
```

is replaced with an sql_exec predicate:

```
sql_exec("select R1,...,Rz from rel where <condition>",
  B1,...,By,F1,...,Fz)
```

B_1 to B_y are those variables of W_1 to W_n which are bound. The <condition> clause is a conjunction of equality tests:

```
S1=!v1 and ... and Sy=!vy
```

where S_1 to S_y are the names of the attributes in rel which correspond to B_1 to B_y. $!v_1$ to $!v_y$ are variables which will be replaced with values at execution time.

In the example above, there is one bound variable; NAME. The corresponding attribute in the employee relation is name. Hence, the <condition> clause is

```
name=!v1
```

and B_1 to B_y is

```
NAME
```

F_1 to F_z are those variables of W_1 to W_n which are unbound. R_1 to R_z are the names of the attributes in rel which correspond to those variables. In the example above, there are two unbound variables; ENR and SAL. The corresponding attributes in the employee relation are enr and salary. Hence, R_1 to R_z is

```
enr,salary
```

and F_1 to F_z is

```
ENR,SAL
```

Limitations of the naive algorithm

A serious drawback of the naive algorithm is that joins that could be performed in the relational database are performed in the Translator. For

example, consider the query

```
select district(s),salary(s) for each salesman s
```

This will be translated to the following TR rule:

```
temp_{charstring,integer}(D,SAL)
                  <- r_salesman(ENR,D,_) &
                     r_employee(ENR,_,SAL,_)
```

If the naive algorithm was used, the executable TBR program would have two sql_exec predicates:

```
temp^{ff}_{charstring,integer}(D,SAL)
                  <- sql_exec^{bff}_{charstr.,integer,charstr.}("select
                       enr,district from salesman",ENR,D) &
                     sql_exec^{bbf}_{charstring,integer,integer}("select
                       salary from employee where enr=!v1",
                       ENR,SAL)
```

Since the first query would return two tuples (there are two salesmen in the database), the second query would be executed two times, for a total of three calls to the relational database. The join between the salesman and employee tables is performed in the Translator.

An optimal TBR program should have only one sql_exec predicate:

```
temp^{ff}_{charstring,integer}(D,SAL)
                  <- sql_exec^{bff}_{charstr.,charstr.,integer}("select
                       district,salary from salesman,
                       employee where salesman.enr=
                       employee.enr",D,SAL)
```

This would result in a single query to the relational database. The join between the salesman and employee tables is performed in the relational database.

Another problem with the naive algorithm is that some tests that could be performed in the relational database are performed in the Translator. This leads to unnecessary shipment of data between the relational database and the Translator. For example, consider the query:

```
select name(e)
for each employee e
where salary(e)>18000
```

This will be translated to the following TR rule:

```
temp_{charstring}(N)  <- r_employee(_,N,SAL,_) &
                         >_{integer,integer}(SAL,18000)
```

If the naive algorithm is used, the executable TBR program would be:

```
temp^{f}_{charstring}(N)  <- sql_exec^{bff}_{charstr.,charstr.,integer}("select
                             name,salary from employee",N,SAL) &
                           >^{bb}_{integer,integer}(SAL,18000)
```

The query would return all names and salaries. The salaries would then be checked by the Translator to see what names should be returned.

An optimal TBR program would perform the salary check in the relational database:

```
temp^f_charstring(N)  <- sql_exec^bf_charstring,charstring("select
                           name from employee where
                           salary>18000",N)
```

Since only one employee earns more than 18000, the second query would return a single tuple whereas the first query would return all four tuples in the employee table.

Merging r-predicates
Consider again the following TR rule:

```
temp_charstring,integer(D,SAL)
                 <- r_salesman(ENR,D,_) &
                    r_employee(ENR,_,SAL,_)
```

Three possible execution strategies are possible:

```
temp_charstring,integer(D,SAL)
                 <- r_salesman(ENR,D,_) &
                    r_employee(ENR,_,SAL,_)
temp_charstring,integer(D,SAL)
                 <- r_employee(ENR,_,SAL,_) &
                    r_salesman(ENR,D,_)
temp_charstring,integer(D,SAL)
                 <- { r_salesman(ENR,D,_)
                      r_employee(ENR,_,SAL,_) }
```

The first two strategies result in two sql_exec predicates. In the third strategy the two predicates are merged. Merged predicates are placed within brackets. This will result in a single sql_exec predicate:

```
sql_exec^bff_charstring,charstring,integer("select district,salary
  from salesman,employee where salesman.enr=employee.enr",
  D,SAL)
```

In this example it is obvious that the third execution strategy (merging the r-predicates) is optimal. The reason for this is that the variable ENR is present in both predicates. This means that the resulting SQL query will involve a join on the enr attributes of the salesman and employee relations.

It is not always optimal to merge all r-predicates. Recall that a Translator is a complete AMOS DBMS which is extended with the notions of mapped types and mapped objects. This means that the schema of a Translator will contain both functions for which the extension is stored directly in the Translator and functions for which the extension depends on the state of the relational database. This means that some queries will result in query plans

where there are r-predicates which have no common variables. Merging these predicates will result in a 'cartesian product query' against the relational database, rather than a 'join query'. This may sometimes be desirable, sometimes not. To illustrate this, we will assume that the Translator schema in the company example is enriched with the following function:

```
recreation(charstring)->charstring
```

The function takes a district as argument and returns the recreational activities that are possible in that district. The extension is stored directly in the Translator and is illustrated in table 3.

district	activity
'kisa'	'squash'
'kisa'	'fishing'
'kisa'	'skiing'
'rimforsa'	'squash'

Table 3: The extension of the function *recreation(charstring district)->charstring activity.*

Now suppose that the following query is given to the Translator ('what salesmen work in a district where the hobby of some employee can be practiced'):

```
select s
for each salesman s, employee e
where recreation(district(s))=hobby(e)
```

The TR rule for this query will be:

$$\texttt{temp}_{\texttt{salesman}}\texttt{(S)} \quad \texttt{<- oidmap1}_{\texttt{employee,integer}}\texttt{(S,ENR) \&}$$
$$\texttt{r_salesman(ENR,D,_) \&}$$
$$\texttt{recreation}_{\texttt{charstring,charstring}}\texttt{(D,A) \&}$$
$$\texttt{r_emp_hobbies(_,A)}$$

There are 22 ways to execute this ObjectLog program (16 where the r-predicates are not merged and 6 where they are). We will show two representative examples of these. The execution trees for these examples are shown in figure 31.

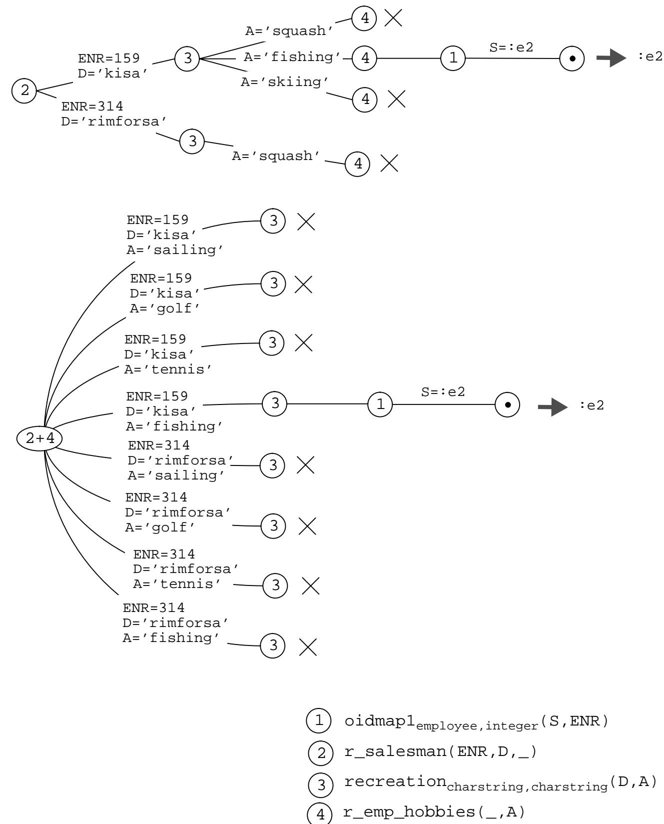


Figure 31: Execution trees for the two example execution plans.

Example 1:

$$\texttt{temp}^{f}_{salesman}\texttt{(S)} \leftarrow \texttt{r_salesman(ENR,D,_)} \ \& \ \texttt{recreation}^{bf}_{charstring,charstring}\texttt{(D,A)} \ \& \ \texttt{r_emp_hobbies(_,A)} \ \& \ \texttt{oidmap1}^{fb}_{employee,integer}\texttt{(S,ENR)}$$

The r_salesman predicate will result in the execution of the SQL query

```
select enr,district from salesman
```

which will return two tuples. The recreation predicate will be executed

twice, once for each district that was returned by the r_salesman predicate. The r_emp_hobbies predicate will result in execution of queries on the form

```
select T from emp_hobbies where hobby=!v1
```

where !v1 is substituted with the four activities that was returned by the recreation predicate. This gives a total of five queries against the relational database.

Example 2:

$$\texttt{temp}^{f}_{salesman}\texttt{(S)} \quad \texttt{<- \{ r_salesman(ENR,D,_)}$$
$$\texttt{r_emp_hobbies(_,A) \} \&}$$
$$\texttt{recreation}^{bb}_{charstring,charstring}\texttt{(D,A) \&}$$
$$\texttt{oidmap1}^{fb}_{employee,integer}\texttt{(S,ENR)}$$

The first predicate will result in execution of the query

```
select enr,district,hobby from salesman,emp_hobbies
```

which will return eight tuples (the cartesian product of the salesman and emp_hobbies relations).[63] This means that the recreation predicate is executed eight times. However, since the execution plan only involves a single query to the relational database, it is not obvious that this plan is inferior to the first one. It depends on the relative cost of computations in the Translator, transmission costs, and the costs of processing the queries in the relational database.

To achieve an accurate estimation of the costs of the relational queries, the cost model of the relational database system will have to be simulated in the Translator. Note that the statistics and cost model parameters of the relational database system may not be available. In that case they could be estimated by running a well-chosen set of test queries [22].

Semantics of the sql_exec predicate

There is only one resolvent predicate for sql_exec:

$$\texttt{sql_exec}^{*}_{*}\texttt{(QUERY,\&REST ARGLIST)}$$

The stars (*) used in the signature instead of a binding pattern and a type list indicate that this resolvent is used for all binding patterns and for all types of the arguments. The predicate can have a varying number of arguments.[64]

The semantics of the sql_exec resolvent predicate is as follows:

Let y be the number of arguments that are bound.[65] Then, let boundlist (B_1, ..., B_y) be the first y members of ARGLIST, and let freelist (F_1, ..., F_z) be

63. In most relational database systems, queries return *bags* of tuples rather than *sets* of tuples (as relational database theory prescribes). If this is the case, the query would return 14 tuples rather than 8.

64. Hence the &REST form of the second argument (CommonLisp style). ARGLIST is a list of the rest of the arguments.

the rest of ARGLIST. Substitute each occurrence of $!v_X$ in QUERY with the value of the Xth member of boundlist. Execute the substituted query in the relational database. For each result tuple <R_1, ..., R_z>, let (QUERY, B_1, ..., B_y, R_1, ..., R_z) be a result binding to (QUERY, B_1, ..., B_y, F_1, ..., F_z).

13.4.4 Example

We conclude this section with an example. The example query is the same query which was used in section 12.3 to illustrate the query processing steps for normal AMOS databases. The difference here is that the AMOS schema is a view of a relational database. The extension is not stored directly in the AMOS database.

```
select s, doublesalary(manager(s))
for each salesman s
where hobby(s)='golf'
```

The query is translated to a TR rule in the same way as in section 12:

```
temp_{salesman,integer}(S,DS)
                 <- salary_{employee,integer}(M,SAL) &
                    times_{integer,integer,integer}(SAL,2,DS) &
                    manager_{employee,employee}(S,M) &
                    hobby_{employee,charstring}(S,'golf') &
                    typesof_{object,type}(S,:typeSalesman)
```

This time, however, salary, manager, and hobby are derived functions. The TR rules for these functions are:

```
salary_{employee,integer}(E,SAL)
                 <- oidmap1_{employee,integer}(E,ENR) &
                    r_employee(ENR,_,SAL,_)
manager_{employee,employee}(E,M)
                 <- oidmap1_{employee,integer}(E,ENR) &
                    oidmap1_{employee,integer}(M,MNR) &
                    r_employee(ENR,_,_,MNR)
hobby_{employee,charstring}(E,H)
                 <- oidmap1_{employee,integer}(E,ENR) &
                    r_emp_hobbies(ENR,H)
```

This means that the expanded TR rule for the query is:

---

65. This can be derived by looking at the result specification part of the query. If the query returns tuples of arity n (i.e. the query is 'select R_1, ..., R_n from ... where ...'), then the predicate has n unbound arguments (and hence len-n bound arguments, where len is the length of ARGLIST). In the actual implementation, the number of bound arguments is supplied as an extra argument to sql_exec, to avoid calculation of this at run-time.

```
temp_salesman,integer(S,DS)
                  <- oidmap1_employee,integer(M,MNR) &
                     r_employee(MNR,_,SAL,_)
                     times_integer,integer,integer(SAL,2,DS) &
                     oidmap1_employee,integer(S,SNR) &
                     oidmap1_employee,integer(M,MNR2) &
                     r_employee(SNR,_,_,MNR2)
                     oidmap1_employee,integer(S,SNR2) &
                     r_emp_hobbies(SNR2,'golf')
                     typesof_object,type(S,:typeSalesman)
```

Since the second argument to the typesof predicate is a constant, the following holds:

```
typesof_object,type(S,:typeSalesman)
                  <- oidmap1_employee,integer(S,SNR) &
                     r_salesman(SNR,_)
```

The output of the typesof Expander is:

```
temp_salesman,integer(S,DS)
                  <- oidmap1_employee,integer(M,MNR) &
                     r_employee(MNR,_,SAL,_)
                     times_integer,integer,integer(SAL,2,DS) &
                     oidmap1_employee,integer(S,SNR) &
                     oidmap1_employee,integer(M,MNR2) &
                     r_employee(SNR,_,_,MNR2)
                     oidmap1_employee,integer(S,SNR2) &
                     r_emp_hobbies(SNR2,'golf')
                     oidmap1_employee,integer(S,SNR3) &
                     r_salesman(SNR3,_)
```

Compile-time unification gives:

```
temp_salesman,integer(S,SAL)
                  <- oidmap1_employee,integer(M,MNR) &
                     r_employee(MNR,_,SAL,_) &
                     times_integer,integer,integer(SAL,2,DS) &
                     oidmap1_employee,integer(S,SNR) &
                     r_employee(SNR,_,_,MNR) &
                     r_emp_hobbies(SNR,'golf') &
                     r_salesman(SNR,_)
```

Removal of unnecessary oidmap1 predicates gives the final TR rule:

```
temp_salesman,integer(S,SAL)
                  <- r_employee(MNR,_,SAL,_) &
                     times_integer,integer,integer(SAL,2,DS) &
                     oidmap1_employee,integer(S,SNR) &
                     r_employee(SNR,_,_,MNR) &
```

```
                     r_emp_hobbies(SNR,'golf') &
                     r_salesman(SNR,_)
```

In this case, it is optimal to merge all the r-predicates together with the times predicate. The optimized TBR rule is:[66]

$\texttt{temp}^{ff}_{salesman,integer}$(S,SAL)

```
                  <- { r_employee(MNR,_,SAL,_)
                       times_integer,integer,integer(SAL,2,DS)
                       r_employee(SNR,_,_,MNR)
                       r_emp_hobbies(SNR,'golf')
                       r_salesman(SNR,_) } &
                     oidmap1^fb_employee,integer(S,SNR)
```

The executable TBR rule, where the merged r-predicates and times predicate have been replaced with an sql_exec predicate, is:

$\texttt{temp}^{ff}_{salesman,integer}$(S,SAL)

```
                  <- sql_exec*_*("select e1.enr, 2*e2.salary
                       from employee e1 e2, emp_hobbies,
                       salesman where hobby='golf' and
                       employee=e1.enr and e1.enr=
                       salesman.enr and e1.manager=e2.enr",
                       SNR,SAL) &
                     oidmap1^fb_employee,integer(S,SNR)
```

66. This is not implemented in the current prototype. The naive algorithm which is used presently would perform the join as well as the multiplication in the Translator rather than in the relational database. See section 13.4.3.

14 Summary of Part II

An object view of relational data makes it possible to transparently work with data in a relational database as if the data were stored in an object-oriented database. Queries against the object view are translated to queries against the relational database. The results of these queries are then processed to form an answer to the initial query. We have developed such a view mechanism and the second part of thesis described the principles behind its implementation.

We discussed the relationship between schemas in object-oriented data models and the corresponding relational schemas. A normal form, SSNF, for representing subtype/supertype relationships in relational schemas was introduced. By having the relational database schema in SSNF, the mapping between the object view and the relational database is greatly simplified. When an object view is created over a relational database that is not in SSNF, the first step is to define a relational view that is.

The term Translator was used for the software which implements object views of relational data. A Translator is a complete AMOS DBMS augmented with the notions of mapped types and mapped objects. A mapped type is a type for which the extension is defined in terms of the state of a relational database. When the relational database is in SSNF, there is a one-to-one mapping between instances of a mapped type and tuples in some relation in the relational database. The instances of mapped types are called mapped objects.

ObjectLog, an object-oriented logical language, is used for the internal representation of query plans in Translators. We showed the importance of having all relational database access explicitly represented in Translator query plans. The use of r-functions and r-predicates was introduced as a way to handle this.

Object identity is provided in the view through the use of mapping tables. OIDs are generated dynamically the first time they are needed and are thereafter maintained by the Translator. The mapping between OIDs and primary key values is modelled with special kinds of functions called oidmap functions. We discussed OID management during query processing and how it can be performed without embedding relational database access in the code of the oidmap functions.

In an object view of a relational database, the instance-of relationship (which objects are instances of which types) is dependent on the state of the relational database. This means that type membership tests require access to the relational database. We discussed the role of type membership tests during query processing and the problems of generating optimal execution

strategies for queries involving these tests.

The optimizer finds the optimal execution strategy by estimating the cost for the different reorderings of the predicates in the query plan. The use of r-predicates requires new query optimization techniques. r-predicates represent relational database access but are not executable as they stand. After query optimization, they are replaced by executable sql_exec predicates which make the calls to the relational database. The current implementation of Translators uses a naive algorithm for this which replaces each r-predicate with an sql_exec predicate. An optimal algorithm should sometimes replace multiple r-predicates by a single sql_exec predicate. This depends on the relative costs of computations in the Translator, transmission costs, and the cost of processing the queries in the relational database.

Prior to query optimization, two techniques can be used to simplify the query plan. The technique of compile-time unification uses information about key attributes to replace multiple predicates with a single one. The technique of removal of unnecessary predicates can be applied to a certain class of predicates which only serve as generators of OIDs.

Concluding Remarks

We refer to section 7 and section 14 for summaries of the two parts of the thesis and to the Preface for a discussion on the contributions of this work.

The survey of multidatabase system architectures and the discussion on the role of the canonical data model should make it clear that object views of different kinds of data sources is a central issue in multidatabase systems. The popularity of the relational data model makes object views of relational data particularly important.

We have presented the main problems when object views of relational databases are created and have shown how some of them can be solved.

Object views of relational data is a research area where a lot of work remains to be done. In particular, we are not aware of any research on query processing techniques for such views prior to this work. The following is a description of some of the research problems that are not addressed in this thesis.

Future work

View Definition Language. A formal, simple, and semi-automatic way to define object views of relational data is needed. A formal Object View Definition Language (OVDL) should be developed for this purpose. In the current prototype, the mapping between the relational database and the object view is established by handcoding the effects of imagined OVDL statements.

Unusual relational schemas. As discussed in section 11.3, the first step of the mapping procedure is to define a relational view that is in a certain normal form. Unfortunately, current relational view definition languages are not general enough to allow all kinds of mappings [39] [45]. Two different solutions are possible: (a) Extend relational view definition languages so that a view in the normal form can be defined for all kinds of relational schemas, or (b) Add constructs to the Object View Definition Language so that all relational schemas can be directly mapped to the desired object view, including those for which a relational view in the normal form can not be defined.

Multiple inheritance. The current prototype only handles simple inheritance - types in the object view can only have one other type as a direct supertype. Formalisms and algorithms need to be developed to map relational data to multiple inheritance structures in the object view. The normal form discussed in section 11.3 must be extended to handle cases of multiple inheritance.

Query optimization. More work is needed on the query optimization techniques.

As discussed in section 13.4.3, a rather naive algorithm is used in the current prototype for replacing r-predicates with executable sql_exec predicates. Sometimes it is advantageous to replace multiple r-predicates with a single sql_exec predicate. To find the most effective execution strategy, the optimizer should consider local processing costs, transmission costs, and the cost to execute SQL queries in the relational database. This means that the cost model of the relational database must be simulated in the query optimizer of the Translator [22].

We have emphasized the importance of having all access to the relational database explicitly represented in query plans. As discussed in section 13.3.1, our query processing technique can not do this for all kinds of type membership tests. In rare cases this results in non-optimal execution strategies.

References

[1] S.Abiteboul, A.Bonner: ‘Objects and Views’, *Proc. ACM SIGMOD Conf.*, 1991.

[2] R.Ahmed, P.De Smedt, W.Du, W.Kent, M.A.Ketabchi, W.Litwin, A.Rafii, M-C.Shan: ‘The Pegasus Heterogeneous Multidatabase System’, *IEEE Computer*, Vol. 24, No. 12, December 1991.

[3] R.Ahmed, J.Albert, W.Du, W.Kent, W.Litwin, M-C.Shan: ’An Overview of Pegasus’, *Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS ’9*3, Vienna, Austria, April 1993.

[4] J.Albert, R.Ahmed, M.Ketabchi, W.Kent, M-C.Shan: ’Automatic Importation of Relational Schemas in Pegasus’, *Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS ’9*3, Vienna, Austria, April 1993.

[5] M.Atkinson et al: ‘The Object-Oriented Database System Manifesto’, *Proc. of the 1st Intl. Conf. on Deductive and Object-Oriented Databases*, Kyoto, Japan, December 1989.

[6] C.Batini, M.Lenzerini, S.B.Navathe: ‘A Comparative Analysis of Methodologies for Database Schema Integration’, *ACM Computing Surveys*, Vol. 18, No. 4, December 1986.

[7] D.Beech: ’Collections of Objects in SQL3’, *Proc. VLDB ’93*, Dublin, Ireland, 1993.

[8] E.Bertino, L.Martino: ‘Object-Oriented Database Management Systems: Concepts and Issues’, *IEEE Computer*, Vol. 24, No. 4, April 1991.

[9] E.Bertino: ’A View Mechanism for Object-Oriented Databases’, *Proc. 3rd Intl. Conf. on Extending Database Technology, EDBT ’92* (Lecture Notes in Computer Science 580, Springer-Verlag), March 1992.

[10] A.Bouguettaya, R.King: ’Large Multidatabases: Issues and Directions’, *Proceedings of the Conference on Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, November 1992.

[11] M.W.Bright, A.R.Hurson, S.H.Pakzad: ‘A Taxonomy and Current Issues in Multidatabase Systems’, *IEEE Computer*, Vol. 25, No. 3, March 1992.

[12] M.L.Brodie, M.Stonebraker: ’DARWIN: On the Incremental Migration of Legacy Information Systems’, TR-0222-10-92-165, GTE

Laboratories, March 1993.

[13] A.Cardenas: 'Heterogeneous Distributed Database Management: The HD-DBMS', *Proc. of the IEEE*, Vol. 75, No. 5, May 1987.

[14] M.Castellanos: 'Semantic Enrichment of Interoperable Databases', *Proceedings of the Workshop on Interoperability in Multidatabase Systems, RIDE-IMS '9*3, Vienna, Austria, April 1993.

[15] R.G.G.Cattell: *Object Data Management*, Addison-Wesley Publishing Company, 1992.

[16] R.G.G.Cattell (ed.): *The Object Database Standard: ODMG-93*, Morgan Kaufmann Publishers, San Mateo, California, 1994.

[17] S.Ceri, G.Gottlob, L.Tanca: *Logic Programming and Databases*, Springer-Verlag, 1990.

[18] J.Chomicki, W.Litwin: 'Declarative Definition of Object-Oriented Multidatabase Mappings', in Ozsu, Dayal, Valduriez (eds): *Distributed Object Management*, Morgan Kaufmann Publishers, 1994.

[19] E.Codd: 'A Relational Model for Large Shared Data Banks', *Communications of the ACM*, Vol. 13, No. 6, June 1970.

[20] C.J.Date: *An Introduction to Database Systems*, Volume I, Fourth Edition, Addison-Wesley Publ. Company, 1986.

[21] B.Demuth, A.Geppert, T.Gorchs: 'Algebraic Query Optimization in the CoOMS Structurally Object-Oriented Database System', in J.C.Freytag et al. (eds.): *Query Processing For Advanced Database Systems*, Morgan Kaufmann Publishers Inc., 1994.

[22] W.Du, R.Krishnamurthy, M-C.Shan: 'Query Optimization in Heterogeneous DBMS', *Proc. VLDB '92*, Vancouver, Canada, 1992.

[23] R.Elmasri, S.B.Navathe: *Fundamentals of Database Systems*, The Benjamin/Cummings Publishing Company, Inc., 1989.

[24] G.Fahl, T.Risch, M.Sköld: 'AMOS - An Architecture for Active Mediators', *Proc. Intl. Workshop on Next Generation Information Technologies and Systems, NGITS '93*, Haifa, Israel, June 1993.

[25] P.Fankhauser, W.Litwin, E.Neuhold, M.Schrefl: 'Global View Definition and Multidatabase Languages - Two Approaches to Database Integration', in R.Speth (ed.): *Research into Networks and Distributed Applications*, Elsevier Science Publ., North-Holland, 1988.

[26] D.H.Fishman et al: 'Overview of the Iris DBMS', in W.Kim, F.H.Lochovsky (eds.): *Object-Oriented Concepts, Databases and Applications*, ACM Press, Addison-Wesley, 1989.

[27] E.N.Hanson: 'A Performance Analysis of View Materialization

Strategies', *Proc. ACM SIGMOD Conf.*, May 1987.

[28] E.N.Hanson, J.Widom: 'An Overview of Production Rules in Database Systems', *The Knowledge Engineering Review*, Vol. 8, No. 2, 1993.

[29] S.Heiler, S.Zdonik: 'Object Views: Extending the Vision', *Proc. of the 6th Intl. Conf. on Data Engineering*, L.A., California, USA, February 1990.

[30] D.Heimbigner, D.McLeod: 'A Federated Architecture for Information Management', *ACM Transactions on Office Information Systems*, Vol. 3, No. 3, July 1985.

[31] D.Hsiao: 'Tutorial on Federated Databases and Systems (Part I)', *The VLDB Journal*, Vol. 1, No. 1, July 1992.

[32] R.Hull, R.King: 'Semantic Database Modeling: Survey, Applications, and Research Issues', *ACM Computing Surveys*, Vol. 19, No. 3, September 1987.

[33] Y.E.Ioannidis, Y.C.Kang: 'Randomized Algorithms for Optimizing Large Join Queries', *Proc. ACM SIGMOD Conf.*, Atlantic City, 1990.

[34] P.Johannesson, K.Kalman: 'A Method for Translating Relational Schemas into Conceptual Schemas', *Intl. Conf. on Entity-Relationship Approach*, Toronto, 1989.

[35] W.Kent: 'The many forms of a single fact', *Proc. IEEE Compcon 89*, San Francisco, February 1989.

[36] R.L.Kernighan: 'Are You Ready for ODBC?', *DBMS Magazine*, October 1992.

[37] W.Kim, E.Bertino, J.F.Garza: 'Composite Objects Revisited', *Proc. ACM SIGMOD Conf.*, 1989.

[38] R.Krishnamurthy, C.Zaniolo: 'Optimization in a Logic Based Language for Knowledge and Data Intensive Applications', *Proc. Intl. Conf. on Extending Database Technology, EDBT '88*, Venice, Italy, 1988.

[39] R.Krishnamurthy, W.Litwin, W.Kent: 'Language Features for Interoperability of Databases with Schematic Discrepancies', *Proc. ACM SIGMOD Conf.*, 1991.

[40] T.Landers, R.Rosenberg: 'An Overview of Multibase', in H-J.Schneider (ed.): *Distributed Databases*, North-Holland, 1982.

[41] K.H.Law, T.Barsalou, G.Wiederhold: 'Management of Complex Structural Engineering Objects in a Relational Framework', in *A Mediator Architecture for Abstract Data Access*, Research Report

STAN-CS-90-1303, Stanford University, California, February 1990.

[42] D.Linthicum: 'Moving Away From the Network, Using Middleware', *DBMS Magazine*, Vol. 7, No. 1, January 1994.

[43] W.Litwin, A.Abdellatif: 'Multidatabase Interoperability', *IEEE Computer*, Vol. 19, No. 12, December 1986.

[44] W.Litwin, L.Mark, N.Roussopoulos: 'Interoperability of Multiple Autonomous Databases', *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[45] W.Litwin, M.Ketabchi, R.Krishnamurthy: 'First Order Normal Form for Relational Databases and Multidatabases', *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[46] W.Litwin: 'O*SQL: A Language for Object Oriented Multidatabase Interoperability', *Proceedings of the Conference on Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, November 1992.

[47] W.Litwin, T.Risch: 'Main Memory Oriented Optimization of OO Queries Using Typed Datalog with Foreign Predicates', *IEEE Transactions on Knowledge and Data Engineering*, Vol. 4, No. 6, December 1992.

[48] P.Lyngbaek et al: 'OSQL: A Language for Object Databases', Technical Report, HP Labs., HPL-DTD-91-4, January 1991.

[49] V.M.Markowitz, J.A.Markowsky: 'Identifying Extended Entity-Relationship Object Structures in Relational Schemas', *IEEE Trans. on Software Engineering*, Vol. 16, No. 8, August 1990.

[50] A.Motro: 'Superviews: Virtual Integration of Multiple Databases', *IEEE Transactions on Software Engineering*, Vol. 13, No. 7, July 1987.

[51] S.B.Navathe, A.M.Awong: 'Abstracting Relational and Hierarchical Data with a Semantic Data Model', *Proc. of the Sixth Intl. Conf. on Entity-Relationship Approach*, New York, USA, November 1987.

[52] M.A.Ouksel, C.F.Naiman: 'Cooperation in Heterogeneous Database Systems', *Proc. Intl. Workshop on Next Generation Information Technologies and Systems, NGITS '93*, Haifa, Israel, June 1993.

[53] M.T.Özsu, P.Valduriez: *Principles of Distributed Database Systems*, Prentice-Hall, 1991.

[54] T.Risch: 'Monitoring Database Objects', *Proc. of the 15th International Conference on Very Large Data Bases, VLDB '89*, Amsterdam, the Netherlands, 1989.

[55] F.Saltor, M.Castellanos, M.Garcia-Solaco: 'Suitability of Data Mod-

els as Canonical Models for Federated Databases', *SIGMOD RECORD*, Vol. 20, No. 4, December 1991.

[56] A.P.Sheth, J.A.Larson: 'Federated Database Systems for Managing Distributed, Heterogeneous and Autonomous Databases', *ACM Computing Surveys*, Vol. 22, No. 3, September 1990.

[57] D.W.Shipman: 'The Functional Data Model and the Data Language DAPLEX', *ACM Transactions on Database Systems*, Vol. 6, No. 1, March 1981.

[58] M.Sköld, T.Risch: 'Compiling Active Object-Relational Rule Conditions into Partially Differentiated Relations', Dagstuhl-Seminar on Active Databases, Germany, March 1994. Available as Research Report LiTH-IDA-R-94-10.

[59] S.Spaccapietra, C.Parent, Y.Dupont: 'Model Independent Assertions for Integration of Heterogeneous Schemas', *VLDB Journal*, Vol. 1, No. 1, July 1992.

[60] D.D.Straube, M.T.Özsu: 'Queries and Query Processing in Object-Oriented Database Systems', *ACM Transactions on Information Systems*, Vol. 8, No. 4, October 1990.

[61] D.Tsichritzis, A.Klug (eds): *The ANSI/X3/SPARC DBMS Framework*, AFIPS Press, 1978.

[62] J.D.Ullman: *Database and Knowledge-Base Systems*, Volume I & II, Computer Science Press, 1988.

[63] J.van den Hoven: 'Plug 'n Play Data: Standards First', *Database Programming & Design*, Vol. 7, No. 1, January 1994.

[64] L-L.Yan, T-W.Ling: 'Translating Relational Schema with Constraints into OODB Schema', *Proceedings of the Conference on Semantics of Interoperable Database Systems*, Lorne, Victoria, Australia, November 1992.

[65] *1993 Database Buyer's Guide Issue*, DBMS Magazine, Vol. 6, No. 7, 1993.

[66] *Buyer's Guide Issue*, Database Programming & Design, Vol. 6, No. 11, 1993.

Index

active database **12, 29, 90**
aggregation **25, 39, 53**
AMOS **29**
AMOS data model **32, 39**
AMOS query processing **68**
AMOS schema architecture **29**
AMOS schemas, graphical notation for **35**
AMOS software components **30**
AMOSQL **36**
attribute **51**
autonomy **3**

backtracking **65**
behaviour **53**
binding pattern **71**

candidate key **51**
canonical data model **5, 12, 13, 23, 39**
class **24**
company example, the **47**
compile-time unification **98**
complex object **25, 39, 53, 55**
component database **3**
component schema **6, 16, 29**
composite object **26**
cost estimation **68, 71**
cost hint function **71**

DAPLEX **32**
DAPLEX semantics **36, 37**
Datalog **64**
deletion semantics **90**
derived function **34, 38, 69**
direct instance **25, 34**
disjoint specialization **25, 58**
distributed database system **3, 11**
distribution **3**

early binding **38**
encapsulation **53**
environment variable **36**
export schema **9, 16**
extension of a function **34, 38**
extension of a relation **51**
extension of a type **32**
external schema **17**

fanout **71**
federated approach **8, 10, 29**
federated database system **15**
federated schema **6, 17, 29**
first normal form **51**
flattened function **69**
Flattener **69**
foreign function **34, 38, 71, 77**
foreign key **51**
function **32, 36, 70**

gateway **20**
generalization **24, 25, 53, 55**
global schema approach **7, 9, 11**
grouping **25, 39, 54**
grouping with order **26, 40, 54**

heterogeneity **3**
heterogeneous database system **15**
heuristic optimization **67**

IDAPI **21**
identity-based data model **80**
impedance mismatch **45**
import schema **9**
inclusion dependency **52, 58**
inheritance **24, 33, 53**
instance **24, 32, 53**
instance by generalization **25**
instance-of relationship **25, 61, 92**
integrated schema **6**
Integrator **31**
integrity constraints **51**
intension of a relation **51**
interoperable database system **15**
IRIS **32**
is-a relationship **24**

key **51**

late binding **38, 70, 92**
legacy system **19, 45**
literal type **32**
local schema **6, 16, 29**
logical representation **64**

mapped object **73, 93**
mapped type **73, 93**
materialized intermediate relation **65**

materialized view **12**
mgmt **73**
mgmt function **74**
middleware **20**
monitoring **90**
most general mapped type **73**
Multibase **31**
multidatabase administration **7**
multidatabase language **9, 30, 39**
multidatabase language approach **9, 10, 29**
multidatabase system **3, 11, 15**
multidatabase system architectures **5**
multi-lingual approach **12**
multiple inheritance **25, 33, 40, 73, 115**
multiple integrated schemas approach **8, 9, 29**
multivalued function **34, 55**
multi-way foreign function **77**

object **24, 32, 53**
object algebra **63**
object identifier **24, 33, 53, 80**
object identity **24, 53, 55, 80**
object view **39**
object view definition language **74, 115**
object views of relational data **45, 73**
ObjectLog **68**
ObjectLog Generator **70**
ObjectLog Optimizer **71**
object-oriented data models **53**
object-oriented database **ii, 63**
object-oriented query language **36, 54**
object-relational database **54**
ObjetLog Interpreter **71**
ODBC **20**
ODMG **54**
OID generation **80**
OID generation, algorithmic **80**
OID mapping tables **81**
oidmap function **83**
oidmap predicate **84**
oidmap table **81, 88**
oidmap1 function **86**
oidmap1 predicate **87**
oidmap1 predicates, removal of **100**
oidmap1 predicates, side effects of **89**
overlapping specialization **25, 58**
overloading **35, 71**

partial specialization **25, 58**
part-of relationship **26, 40**
Pegasus **32, 46, 81**
pipelining **65, 71**
predicate **70**
primary key **51, 58, 80**

query optimization **63, 64, 67, 71, 98**
query plan **63, 64**

randomized optimization **68**
reference schema architecture **16**
referential integrity **51**
relation **51**
relational algebra **52, 64**
relational calculus **52**
relational data model, the **51**
relational database **45, 51**
relational database schema **51**
relational schema **51**
resolution, function name **35, 38**
resolvent **35, 38, 70**
r-function **76, 78**
r-predicate **76, 78, 87**
r-predicates, merging **105**
r-predicates, substitution of **102**

SAG **20**
schema integration **6, 11, 23, 26**
semantic data model **23**
semantic enrichment **23, 32**
semantic expressiveness **23**
semantic heterogeneity **3, 11**
semantic modelling constructs **24, 58**
semantic relativism **26, 40**
signature **34**
snapshot **12**
specialization **24, 25, 53, 55, 58**
SQL **52**
sql_exec predicate **102, 108**
SQL3 **54**
SSNF **60, 73, 81**
state of an object **53**
stored function **34, 38**
subtype/supertype relationship **24, 53, 57, 60**
superkey **51**
surrogate type **32**

TA resolvent **70**
TBR predicate **71**
TBR rule **71**
terminology, multidatabase system **15**
total specialization **25, 58**

TR fact **70**
TR predicate **70**
TR rule **70**
Translator **31, 73**
Translator query processing **98**
tuple **51**
type **24, 32, 53**
Type Checker **70**
type graph **33, 73**
type membership test **70, 92**
typesof Expander **96, 110**
typesof function **92, 93**
typesof predicate **92, 93**

update commands **45**

value-based data model **24, 80**
view **i**