INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

QUERIES ON MUTUALLY NESTED OBJECTS, MOTIVATED BY GIS APPLICATIONS

by

Walid Saliba

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of

Master of Science

School of Computer Science
McGill University, Montreal

February 1997

Abstract

Motivated by Geographical Information Systems (GIS) applications, we introduce a new data model for mutually nested objects. Combining features from relational as well as object-oriented database systems, our data model is efficient for queries involving multiple access patterns. By maintaining symmetrical relationships between entities, we allow nesting to be formulated dynamically at the query level rather than the data model level, thus dissociating the data structure from the access method. In addition, we do not favor one access pattern over another by clustering data in one particular manner, giving therefore flexibility and performance to our system.

In order to integrate nesting into the relational algebra, we propose an extension to Relix, which is an academic database management system. We then show how those modifications can be used in a wide variety of queries, provide an algorithm to translate nested queries into flat relational expressions, and finally show that similar improvements can be applied to SQL allowing nested queries to be expressed more naturally.

Abstrait

Motivés par les systèmes d'information à référence spatiale (SIRS), nous présentons dans cette thèse un nouveau modèle de données pour les objets qui s'incluent mutuellement. Ce modèle regroupe des idées qui proviennent des bases de données relationnelles ainsi que des systèmes orientés-objet. Les relations entre les entités du modèle sont toujours symétriques, ce qui fait que la hiérarchie de la structure est déterminée par la requête et non pas par le modèle lui-même. Ainsi, nous dissocions la méthode d'accès aux données de la structure physique du modèle. En plus, il n'y a aucun regroupement des données selon une méthode d'accès particulière, ce qui rend le modèle flexible et efficace pour les requêtes qui accèdent à la base de données par diverses méthodes d'accès.

Nous intégrons notre modèle dans l'algèbre relationnelles en proposant une extension à Relix, un système académique de gestion de bases de données développé à l'université McGill. Nous montrons ensuite comment ces modifications peuvent être utilisées pour exprimer un grand nombre de requêtes hiérarchiques. Nous fournissons en plus un algorithme qui traduit les requêtes hiérarchiques en expressions relationnelles simples. Ces mêmes changements sont ensuite introduits à SQL, ce qui nous permet d'exprimer les requêtes hiérarchiques de façon plus simple et naturelle que le SQL standard.

Table of Contents

List of Tables

List of Figures

Acknowledgment

I would like to express my sincere gratitude to my supervisor, Professor Timothy Merrett, for his guidance, his extraordinary patience, and his continuous motivation and reassurance. He helped me reach beyond my personal limits and for that I am very thankful.

Many thanks go also to my friends who contributed to this thesis. Isabella Adornato, who helped me generously in the editing, and took time off her vacation to assist me; Elie Saadé, for the fun time we had together while studying for our degrees; Manon Breton, for her love, understanding and enormous support throughout the course of my work.

Most of all, I am very grateful to my parents, who taught me the virtues of knowledge and education. This thesis is my way of saying Thank You.

1. Introduction

1.1 Motivation

The motivation behind this thesis emerges from our interest in the field of Geographical Information Systems (GIS). Working with GIS under a relational database system, we felt that certain relationships were difficult to model efficiently and naturally. For example, if we think of rivers and municipalities as complex objects, they both mutually include each other; a river may run through several municipalities, and each municipality can in turn contain several rivers. This type of symmetrical relationship between entities has not been thoroughly investigated in the context of nested queries. Hence, we decided to design a data model suitable for this type of *mutually nested objects* and which would accommodate the following additional requirements:

- many-to-many relationships between objects;
- an object may be shared by an unlimited number of complex objects;
- an object can at the same time be independent as well as embedded in another object;

We have approached the construction of our data model from an information retrieval perspective. Based on our experience in the field of GIS, we started collecting typical queries from various applications. After a while, we realized that there was a common behavior among all GIS queries we investigated: The access pattern in a given query was imposed dynamically by the query rather than by the data model.

For example, when we ask for restaurants in a given municipality, we first locate the municipality and then search for corresponding restaurants. On the other hand,

if we need to find municipalities that do not have a McDonald, we reverse the order of the search, regardless of the data structure. Since clustering objects in a single way can only favor the performance of one access pattern over others, we decided to pursue a data model which allows *dynamic object definition*. That, we believe, achieves better overall performance for applications with multiple access patterns, such as GIS. Therefore, we decided that the structure of the data representation should be orthogonal to the formulation of the query, and symmetrical with respect to the related entities.

The objective of this thesis is to derive a data model for complex objects which answers to the above stated requirements. Based on that model, we would like to formulate nested queries using a proposed extension to an existing academic database management system. We will not actually build neither the data model nor the extensions to the database language, but will provide the reader with an algorithm for a parser that would read the nested query and translate it into a series of flat relational expressions. Finally, we will attempt to integrate our model and the language extensions to a commercial database language, namely SQL.

1.2 Literature Survey

Work on extending the relational data model started shortly after Codd [1] first introduced the relational model in 1979. Researchers interested in that field at the time agreed that the relational model and the relational algebra were not suitable for engineering and CAD applications. Codd's first normal form was too restrictive for the emerging applications of database systems. Although many of them used engineering applications to justify the much needed extension, very few of them actually used a CAD or an engineering application to build their case.

The scope of this thesis does not allow us to make a full survey of the literature on nested relations and complex objects. However, we can distinguish two major

approaches to overcoming the limitations of the relational model, namely the *non-first-normal-form model* and the *semantic model*. In the remainder of this section, we will review the major influence of each of those schools on our work.

The pioneer of the non-first-normal-form data models is Makinouchi [2]. He proposed to generalize the relational model by relaxing the first normal form proposed by Codd. He also proved that in this case that the second and third normal forms still apply. In his model, however, relationships between objects were hierarchical, and once a relation was nested, it became completely dependent on the parent relation.

Following the footsteps of Makinouchi, Jaeschke and Schek [3] introduced the concept of relational attributes in their NF^2 data model, and added two restructuring operators, namely NEST and UNNEST. The NEST operator forms out of a set of attributes a relation and makes it embedded in the original relation. The UNNEST operator extracts a nested relation one level up. Thomas and Fisher [4] expanded the number of arguments to the NEST and UNNEST operators from one attribute to an unlimited number. They also allowed nested relations of arbitrary but fixed length.

For a certain period of time, NEST and UNNEST were the corner stone of the non-first-normal-form approach. These operators, however, were not what we call well-behaved. For one thing, NEST is not the inverse of UNNEST [8], so a NEST followed by an UNNEST might lead to a different relation than the original one. Second, The NEST operator is not associative [8], so the nested relations may have alternative representations depending on the order of the nest operations [5]. In order to restrain the damage of the NEST and UNNEST operators, several authors imposed restriction on the structure of the nested relations.

For Roth et al. [6, 7, 8], for example, tuples had to be functionally dependent on the atomic attributes of the relation, and they named that structure the *partitioned normal form* (PNF). They also defined a relational calculus, a relational algebra

and restricted them to relations that are in PNF. That way, there was always a sequence of NEST operations that will be an inverse for any sequence of valid UNNEST operations.

Several authors were more concerned with expanding the NF^2 model described in [9,10], such as Pistor and Andersen [11] who allowed atomic attributes as well as lists, multisets and tuples in an orthogonal fashion. Others invested in implementing database management systems based on the non-first-normal-form, such as VERSO [12] and DASDBS [13], or on expanding SQL like query languages [11, 7] to express nested queries according to their models.

The non-first-normal-form data model has contributed enormously to our work. Most important, it showed us that the NEST and UNNEST operators were not the best way to deal with nesting. Using these operators forces the data structure to change according to the way a query is accessing the data, sometimes even causing the loss of information in the database. Besides, their ill behavior imposes too many constraints on the data model. Fortunately, not all pathways created by the non-first-normal-form approach were to be avoided. The nested relational algebra that evolved in this paradigm, for example, is very valuable to our work. The extensions to selection, projection, joins and other relational operators that we find in [7, 8] were an inspiration to our own extension proposed to Relix.

The best known of the semantic data models is probably the Entity-Relationship (ER) model introduce by Chen in 1976 [14]. This paradigm includes other models, however, such as the Functional Data Model (FDM) [15], the Semantic Data Model (SDM) [16], RM/T [17] and IFO [18]. We find in the semantic models the seeds of two important concepts used in the design of our data model, namely the explicit representation of objects and the distinction between organic and composed classes of objects.

The relational model uses tuples to represent entities. Each entity is defined by a unique set of attributes in the relation called a key. Contrary to that philosophy,

semantic models are object-based, which means that objects are represented and identified independently from their associated values. This allows many complex objects to share the same sub-object, and also enables mutual referencing and multiple relationships between objects. This characteristic is manifested in our data model by the use of object identifiers.

Another area where the semantic models had an important but less direct effect on our data model is the definition of complex objects at a conceptual level. The NF^2 approaches created complex objects by physically clustering certain relations within others. On the other hand, the semantic models made a clearer distinction between *base* and *non-base* entities [19]. A base entity corresponds directly to a basic object in the world, while a non-base entity is composed out of base and non-base objects. We have taken this concept even further, allowing an entity to be perceived as organic or composed depending on the type of query in which it is involved.

As can be realised from the above discussion, our work has roots in the relational models as well as the semantic models, which share many traits with the object-oriented paradigm. In fact, in recent years, many researchers have explored this evolutionary path from Relational to Object-Oriented data management systems [20, 21, 22, 23, 24, 25, 26, 27, 28]. For instance, Pirahesh and Lindsay [23] introduced the SQL Extended Normal Form (SQL/XNF) approach, which builds composite objects on top of a RDBMS using views. Those views consist of component tables and the relationships between them.

More recently, Rys et al [29] worked on performance issues related to implementing complex objects over a commercial RDBMS, namely Oracle. Whereas the SQL/XNF approach relies on object sharing, Rys's approach uses data duplication to improve retrieval time, at the expense of an increased cost for update operations. However, by duplicating data on several relational servers,

they are able to achieve intra-transaction parallelism, which reduces the net response time for updates.

Both systems introduce Object-Oriented features into an existing RDBMS. Other approaches consist of either introducing relational features into OODBMSs [30], or making the two coexist and manage the same data[31]. The primary advantage of those hybrid systems is that they combine object-orientation to the widespread and optimised technology of RDBMSs.

1.3 Scope and Outline

The remainder of the thesis is divided into six chapters. In chapter two, we build the case for our data model for complex objects, comparing it to other data representations in the literature. We then use that architecture to model a small hypothetical set of geographical entities.

Chapter three is a brief introduction to Relix, the database management system on top of which we propose to build our data model and the nested relational algebra. We will cover only those topics of Relix that are relevant to our thesis.

In chapter four, we look at the Relix syntax in the context of mutually nested objects. We explain at a conceptual level what are the required modifications to Relix, when to use them, and what impact they have on the end result of the queries. This chapter does not explain to the user how those modifications are handled at the implementation level. Chapter five is fully dedicated for that issue.

Chapter five is the core of our research. It was the starting point in developing our theory and our model. This chapter is a collection of nested queries covering a wide range of nested expressions. Each query is defined in terms of the English statement of the query, the code for the nested expression in Relix, a translation of the nested query into flat Relix expressions, and finally a discussion about the

theme of that query. At the end of chapter five, we present the rules followed by a parser that translates nested queries into flat relational expressions.

Chapter six serves the readers who are more familiar with SQL than Relix. We tried to draw the lessons of chapters four and five and apply them to SQL. On one hand, we propose certain extensions to SQL that would enable us to deal with nested queries. Then, we choose three simple but representative queries from chapter five and express them in what we called nested SQL. Finally, we highlight the advantages nested SQL has over standard SQL.

To conclude the thesis, we present a review of the major contributions our work has offered in the field of complex object modeling. We admit, however, that our work is far from complete, and in that respect suggest various topics that need to be pursued.

2. A Data Structure for Complex Objects

By definition, a complex object is a set of linked objects (simple or complex) which form a single logical entity [32]. In modeling complex objects, we believe that a clear distinction should be made between entities (objects) and the relationships (links) they have with each other. In that regard, our model is compatible with the general semantic model, and in particular with the entity-relationship (ER) model [14] representation. In addition, the relationship between objects should be defined dynamically in the query and not by static model constructs. We argue that this data model is better suited than the nested relational model for applications where the database system is subject to multiple access methods.

As a test field for our model, we picked geographical information as the subject of our study, first because geographical entities are typically complex (nested) and second because of the growing interest in the field of Geographical Information Systems (GIS).

In the first section of this chapter, we will introduce the reader to our conceptual data model for complex objects. In section 2.2, we will compare our model to existing nested models, and finally provide a full example for the modeling of geographical objects in section 2.3.

2.1 The Proposed Structure

One of the main objectives of this research is to accommodate complex objects into the relational model. In particular, we look at complex objects that are *mutually nested.* Such structures were adopted under the name of "mutually recursive objects" by M. Scholl and H.-J. Schek in [20], although their seeds appeared earlier in the literature as "symmetric n:m relationships" [33] and as

"dynamically nested relations" in [34]. They are best explained with the following example.

Given two relations

PARENT (PA_NAME, PA_AGE)

and

CHILD (CH_NAME, CH_AGE)

We could represent the PARENT relation as a complex object defined by the following statement:

PARENT (PA_NAME, PA_AGE, (CHILD)
(CH_NAME, CH_AGE)

where CHILD is embedded in PARENT.

On the other hand, we can equally view CHILD as a complex object with PARENT being the nested part, as shown in the following declaration

CHILD (CH_NAME, CH_AGE, (PARENT)
(PA_NAME, PA_AGE)

Thereafter, CHILD and PARENT relations are called mutually nested. Both of them are independent, stand alone, flat relations, yet one or the other can be transformed into a complex object if we describe it in terms of the other.

In the following sections, we explain how mutually nested relations may be represented in a relational database system. For that purpose, we will introduce two major building blocks of our data model, namely *object identifiers* and *link relations*.

2.1.1 Object Identifiers

In recent years, the object-oriented approach has gained great momentum, bringing valuable concepts to programming languages and database systems. One main feature of object-orientation is *object identity*. Earlier on, object identity gave the edge to the semantic data models [15, 16, 14, 35] over *value-based* extensions to the relational model.

The difference between value-based and object-based systems and a discussion on object identity can be found in [19]. In simple terms, value-based systems identify objects using one or more attribute values called a *key*. Object-based models, on the other hand, use unique identifiers to represent entities independently from their associated attributes and values. So two tuples can have identical attribute values and still be considered different if they have different object identifiers.

Object identifiers (OIDs) can be implemented using surrogates, which are unique values for each object. They are generated internally by the system when an object is created, and destroyed when the object is deleted from the database. They remain invariant and transparent to the user during their life cycle, and cannot be reused for other objects when deleted. In the case of our data model, OIDs are implemented by introducing a system defined attribute to the structure of the flat relation. For example, the schemes of the PARENT and CHILD relations defined above would be augmented as follows:

PARENT (PA_ID, PA_NAME, PA_AGE)

and

CHILD (CH_ID, CH_NAME, CH_AGE)

In order to avoid naming conflicts, the names of the unique identifiers could obey a syntax not allowed for user defined attributes.

One advantage for integrating object identity into database management systems is that objects can be referenced directly. In absence of OIDs, a portion of the object's structure and values must be reproduced in each relation that references it. This is the case for value-based models where relationships between objects can be represented only through foreign keys. Moreover, if the schema of a relation does not include a key, then references to each tuple will require duplication of the structure and the content of that tuple.

The second advantage of object identity is that it nicely represents data sharing and mutual referencing, which are rather cumbersome to represent in a value-based systems [36]. In that sense, OIDs provide natural support for network related objects as opposed to hierarchical structures.

Object identity has also several advantages at the implementation level, in particular its ability to optimize joins [37]. This features is very useful since joins are expensive operations and they are performed repeatedly in the context of nested queries.

2.1.2 Link Relations

The second modeling tool we use to represent mutually nested objects are binary *link relations*. Those are simple binary relations where each tuple represents a link between two objects of the same or different relations. Objects in the link relations are represented by their object identity, reducing hence the size and the structure of the link relations.

A typical link relation for the CHILD and PARENT relations would be as follows:

PACH_LINK (PA_ID, CH_ID)

Again, the name of the link relation is system generated in such a format that it is guaranteed not to raise any conflict with user defined tables in the database.

The main advantage of using link relations is symmetry between the linked objects. Whether we initiate a query from one side of the relationship or the other, the process is equivalent, and the search effort is the same. Hence, we do not favor one access pattern over another. This is coherent with our objective to represent graph related objects rather than hierarchical structures.

Second, link relations are simple to model and easy to implement. Although we have not covered implementation in our research, structures similar to the link relations named *Join indices* have been proposed and tested against other models in the literature [37, 38]. The conclusion was that joins indices were best suited for applications where no particular access pattern dominated the queries. Join indices were proven also to be good join accelerators in the relational model.

Finally, link relations reflect many-to-many relationships and enable object sharing. Along with object identifiers, they are the essential elements of our data model. In the next section, we will show how they can be used on a small scale to model actual data in the CHILD and PARENT relations.

2.1.3 Building Mutually Nested Relations

The CHILD and PARENT relations described above may be represented as two entities linked by the relationship "IS-PARENT-OF/IS-CHILD-OF", as shown in Figure 2-1.

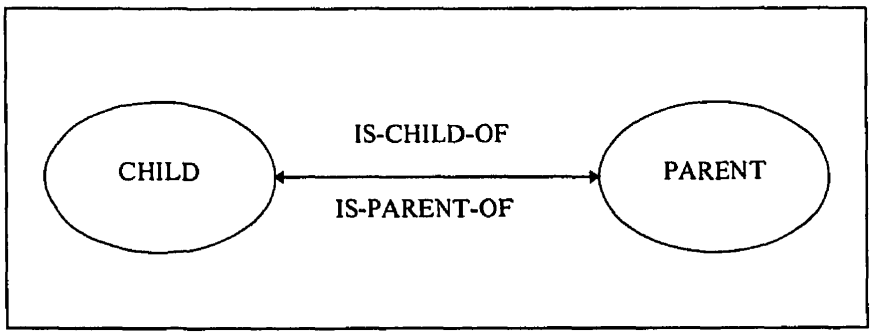


Figure 2-1 Graph representation of mutually nested relations

We will assume that CHILD is composed of the following data:

CH_ID	CH_NAME	CH_AGE
CH1	Sami	12
CH2	Patrick	6
CH3	Kim	9
CH4	Sue	6

Table 2-1 Sample data for the CHILD relation

and that the following are parents

PA_ID	PA_NAME	PA_AGE
PA1	Karim	45
PA2	Peter	29
PA3	Kara	29
PA4	Dana	39

Table 2-2 Sample data for the PARENT relation

Note that the object identifiers in this case are implemented as two characters followed by a sequential number. There is no rule, however, for generating them in the general case, except that they should be globally unique.

We are also given the following parenthood relationships:

- Karim and Kara are Patrick's parents;
- Sue is Dana's daughter;
- Kim and Sami are Peter's children.

The link relation between CHILD and PARENT would therefore look as follows:

PA_ID	CH_ID
PA1	CH2
PA2	CH1
PA2	CH3
PA3	CH2
PA4	CH4

Table 2-3 Link relation for the sample data

2.2 Comparison to Other Representations

In his first paper about nested relations, Makinouchi [2] defines some properties about nested relations. The first of those properties is *dependence*; a tuple from an embedded relation cannot exist without the parent tuple nor independently from it. In other words, the child entity is meaningless without its parent entity. Many authors have also followed that path. Roth states in [8] that "... a particular

nesting scheme should not be used unless functional dependencies that enforce Partitioned Normal Form (PNF) hold in the relation".

We believe that any object has the right to be modeled independently from other entities regardless of what relationships it has with them. The data model should allow the same object to stand alone independently from other objects, to be shared by several complex objects, or to be a composite of other objects, all depending on the access method used in the query. So the nature of the nesting is determined dynamically by the query, and not by a static data representation. Similar objectives were sought by Batory [39] and Mitschang [40] in the Molecular-Atom model (MAD model), although their definition of the complex objects is more explicit.

The main advantage of our approach is symmetry. In static nested relations we have to introduce redundancy in order to achieve symmetry between objects [33], whereas symmetry comes free if we model objects as independent entities related by link relations. Symmetry is revealed in three forms. First, a given sub-object may belong to multiple objects, in other words, it has many parents. For example, a movie theater may belong to a shopping mall, a municipality and to a film company. Second, a sub-object may have many child relations. So several employees can work for a movie theater, and various movies could be playing at the same theater. Finally, a movie theater can exist as an independent entity, without any of its super-objects or sub-objects. The type of relationship an entity assumes is therefore determined by the query and not by the definition of that object.

It is important to add that symmetry cannot exist efficiently without the help of unique object identifiers. Otherwise, the whole key of a relation has to be duplicated each time it is involved in a relationship with another object. The concept of object identifiers is borrowed from object-oriented databases. By introducing it into the relational data model, we hope to close the gap between the

pure relational paradigm on one hand, and the object-oriented paradigm on the other. To our knowledge, the first such effort was made by Scholl and Schek in [20]. Their object algebra, however, is restricted to object preserving operators, i.e. to operators that do not generate new objects. Queries in their model are considered as view definitions on stored objects, whereas our model allows queries to create new objects. In that aspect, our model is closer to the work of Kim [41] and Shaw [42].

Once we have symmetry, we can upgrade the data model from a hierarchical structure into a network representation with possible cyclic structures. In that aspect, our approach differs from those followed in [2, 3, 8, 43, 44] and resembles the semantic data models such as the KL-ONE semantic network [45] and the ER diagram [14]. A positive aspect of network structures is that their physical implementation remains independent of the way data is accessed. Since in hierarchical data models objects are clustered in a single way (without replication), they tend to favor some access patterns at the expense of others. The result is that some queries run very fast while the performance of other syntactically equivalent queries deteriorates. If the application requires a single access pattern, then using our model introduces overhead without any gain. Luckily, however, many applications such as engineering design, CAD tools and GIS all involve multiple access patterns.

Unlike the nested relational approach, objects in our model are always independent and are nested only dynamically within a given query. Hence, we have found no need for restructuring operators such as the NEST and UNNEST operators. This is a big advantage over the other non-first-normal-form relational models, since it frees us from all the drawbacks of NEST and UNNEST, discussed profoundly in [3, 4, 46, 47].

Finally, like most extensions to the relational model, our model fully supports the flat relational approach. In fact, our aim is to have the mutually nested relational

model built on top of flat relational database systems. Unless users choose explicitly to create nesting relationships between objects, relations remain independent just as in the flat model.

2.3 Modeling of Geographical Data

Using the data model defined in section 2.1, we describe in this section the data set on which most examples and queries in the remaining of the thesis will be based. The geographical context is chosen because it lends itself very naturally to complex objects and nested queries. First, we will describe the objects of the data set as stand alone objects, using flat relations. Then, we will create relationships between these objects, and introduce the notion of nesting in the data representation.

The basic elements of the data set are the following:

- A *municipality* is a geographical surface which may be described using the following schema:

  MUNICIPALITY (MU_NAME, MU_MAYOR, MU_POPULATION, MU_AREA)

- A *shopping mall* may be considered either as a polygonal surface, or a point object, depending on the detail level required. It may have the following structure:

  SHOPPING_MALL (SM_NAME, SM_AREA, SM_BUILTIN)

- A *restaurant* is a point object described by the following relation:

  RESTAURANT (RS_NAME, RS_SPECIALITY, RS_CAPACITY, RS_ADDRESS)

- A *movie theater*, also a point object, may have the following attributes in a relation:

  MOVIE_THEATRE (MT_NAME, MT_CAPACITY, MT_COMPANY)

- A *movie* may have many attributes like the title, director, actors, producer and so on. We restrict the movie relation in this data set to the following schema:

  MOVIE (MV_NAME, MV_DIRECTOR, MV_DURATION, MV_YEAR)

The reader might have noticed that we do not store coordinates in our relations. The reason is simply because our interest is not in the spatial representation and handling of geographical entities, but rather in the relationships these objects have with each other. In our case these links happen to be spatial, but that is not restrictive by any means. In fact, the MOVIE relation is not spatial and still obeys the same data model.

The first step in building a complex object model is to introduce object identifiers for the tuples in every relation. Here is a sample data of the new augmented relations:

MUNICIPALITY (MU_ID, MU_NAME, MU_MAYOR, MU_POPULATION, MU_AREA)

MU_ID	MU_NAME	MU_MAYOR	MU_POPULATION	MU_AREA
MU1	Montreal	Richard Bourque	1,000,000	1,000,000
MU2	Longueuil	Patrick Gagnon	300,000	700,000
MU3	London	Philip Cook	1,320,000	800,000
MU4	Brossard	Sue Young	400,000	400,000

Table 2-4 Sample data for the MUNICIPALITY relation

SHOPPING_MALL (SM_ID, SM_NAME, SM_AREA, SM_BUILTIN)

SM_ID	SM_NAME	SM_AREA	SM_BUILTIN
SM1	Eaton	200	1969
SM2	Place Longueuil	270	1986
SM3	La Baie	230	1974
SM4	Champlain	290	1984

Table 2-5 Sample data for the SHOPPING_MALL relation

RESTAURANT (RS_ID, RS_NAME, RS_SPECIALITY, RS_CAPACITY, RS_ADDRESS)

RS_ID	RS_NAME	RS_SPECIALITY	RS_CAPACITY	RS_ADDRESS
RS1	Croissant Plus	Snacks	40	Eaton center
RS2	Le Sultan	Lebanese	200	2354 Laval
RS3	Georgio	Italian	100	827 Tashereau
RS4	Croissant Plus	Snacks	30	Place Longueuil
RS5	Croissant Plus	Snacks	40	1234 Milton
RS6	Tiki Yon	Chinese	60	83 Joliette

Table 2-6 Sample data for the RESTAURANT relation

MOVIE_THEATRE (MT_ID, MT_NAME, MT_CAPACITY, MT_COMPANY)

MT_ID	MT_NAME	MT_CAPACITY	MT_COMPANY
MT1	Loews	2000	Odeon
MT2	Eaton	2550	Famous Players
MT3	Faubourg	1500	Odeon
MT4	Place Longueuil	2000	Odeon
MT5	Cinefun	1800	Famous Players

Table 2-7 Sample data for the MOVIE_THEATRE relation

MOVIE (MV_ID, MV_NAME, MV_DIRECTOR, MV_YEAR, MV_DURATION)

MV_ID	MV_NAME	MV_DIRECTOR	MV_YEAR	MV_DURATION
MV1	Nell	Judie Foster	1995	102
MV2	12 Monkeys	Terry Gilliam	1995	96
MV3	The Silence of the Lambs	Jonathan Demme	1994	113
MV4	Dances with Wolf	Kevin Kostner	1993	92
MV5	Wayne's World	Penelope Spheeris	1994	107

Table 2-8 Sample data for the MOVIE relation

Next, let us describe the initial relationships that exist between those objects.

- The municipality is the main complex objet in this data set. It may include shopping malls, restaurants and movie theaters.
- A shopping mall is a complex object, and can in turn include restaurants and movie theaters.

- A restaurant is an atomic object, contained by a municipality, a shopping mall or both.
- A movie theater is a complex object, contained in a municipality, a shopping mall or both. It may have several theaters, and hence can feature several movies. Although the relation between movie theaters and featured movies is not geographical in nature, it is also a nested relation.
- A movie is an atomic object, and may only be linked to a movie theater.

The relationships between these objects are best described with the following graph:

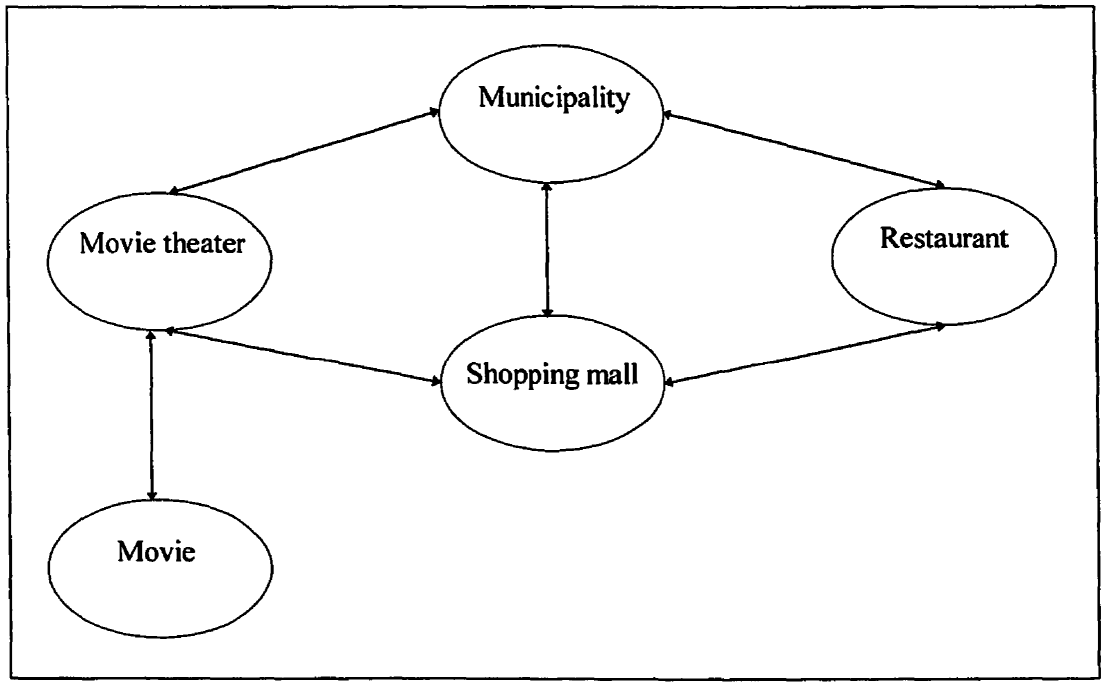


Figure 2-2 Graph showing the links between objects of the data set

In order to model the network structure described in Figure 2-2 , we need, for every double arrowhead connection a link relation. The picture below shows graphically how the geographical objets of the sample data are linked to each

other. As for the movies, we assume that they are showing at the following movie theaters:

- "Nell" and "Dances with Wolf" are showing at LOEWS;
- "12 Monkeys" and "The Silence of the Lambs" are displayed at the Faubourg;
- "Wayne's World" is available at all Famous Players theaters;

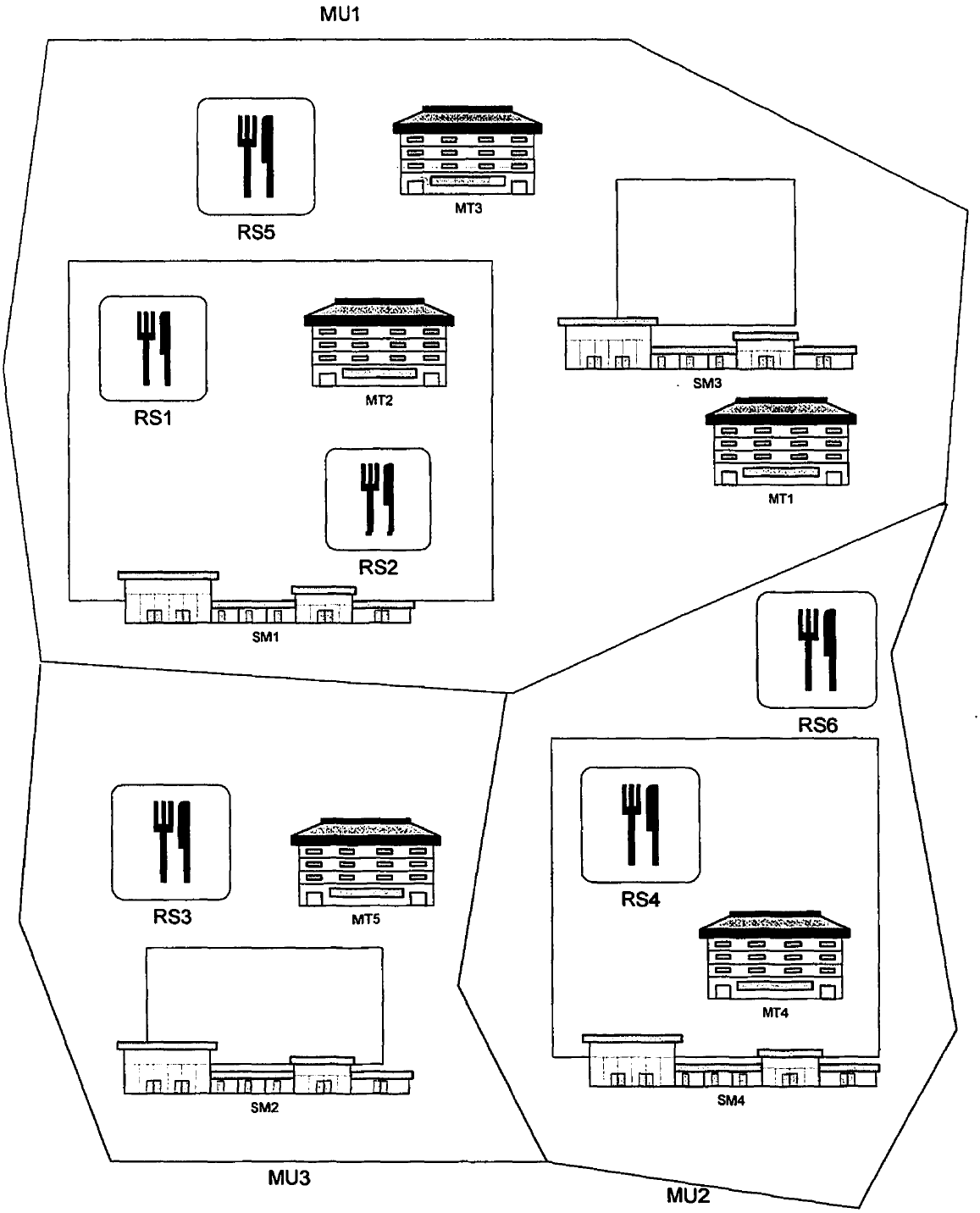


Figure 2-3 Geographical relationship between objets of the sample data

The resulting link relations are described as follows:

- MUSM_LINK (MU_ID, SM_ID)

  Link relation between municipalities and shopping malls;

MU_ID	SM_ID
MU1	SM1
MU1	SM3
MU2	SM4
MU3	SM2

Table 2-9 Link between municipalities and shopping malls

- MUMT_LINK (MU_ID, MT_ID)

  Link relation between municipalities and movie theaters;

MU_ID	MT_ID
MU1	MT1
MU1	MT2
MU1	MT3
MU2	MT4
MU3	MT5

Table 2-10 Link between municipalities and movie theaters

- MURS_LINK (MU_ID, RS_ID)

  Link relation between municipalities and restaurants;

MU_ID	RS_ID
MU1	RS1
MU1	RS2
MU1	RS5
MU2	RS4
MU2	RS6
MU3	RS3

Table 2-11 Link between municipalities and restaurants

- SMMT_LINK (SM_ID, MT_ID)

  Link relation between shopping malls and movie theaters;

SM_ID	MT_ID
SM1	MT2
SM4	MT4

Table 2-12 Link between shopping malls and movie theaters

- SMRS_LINK (SM_ID, RS_ID)

  Link relation between shopping malls and restaurants;

SM_ID	RS_ID
SM1	RS1
SM1	RS2
SM4	RS4

Table 2-13 Link between shopping malls and restaurants

- MTMV_LINK (MT_ID, MV_ID)

  Link relation between movie theaters and movies.

MT_ID	MV_ID
MT1	MV1
MT1	MV4
MT2	MV5
MT3	MV2
MT3	MV3
MT5	MV5

Table 2-14 Link between movies and movie theaters

By introducing the link relations and the object identifiers to the original relations, we have extended the flat relational model into a mutually nested model. Now we are ready to tackle the relational algebra that operates on the new data representation. Since the proposed algebra is an extension to the Relix system, we dedicate the next chapter to presenting those features of Relix that are affected by our work.

3. Relix

3.1 Introduction

This chapter is an introduction to Relix, the language used for the query definition in this work. The name Relix is a concatenation of the abbreviations of Relations and UNIX. The language was originally developed at the school of Computer Science at McGill, in the context of the Aldat project.

The first implementation of Relix produced in 1982 [48] aimed at administrative data processing applications. In the mid and late 80's, Relix was used to manipulate unstructured data such as text and pictures. Today, with features such as procedures, event handling and computations, Relix is evolving from a database system into a programming language for relational database access.

An in depth presentation of Relix can be found in [49]. In this chapter, we will describe the basic concepts of Relix, as well as other topics required for a full understanding of this thesis. Some features of Relix were left out because they were either experimental, had no impact on nesting or were simply judged beyond the scope of this work. The full syntax of Relix can be found in Appendix A.

3.2 Terminology

Relix is mainly composed of a relational algebra and a domain algebra. When it comes to relations, relational algebra plays the same role scalar operators (+, −, / ...) have on numbers. It manipulates relations as a single object, a unit, rather than a set of tuples, exactly as the plus sign (+) hides the operations performed on each digit during an addition.

Domain algebra, as the name indicates, deals more with domains and attributes rather than with the relation itself. In fact, a domain algebra expression holds no reference to the relation name. There are two types of domain algebra expressions, namely horizontal and vertical. The result of a horizontal expression depends uniquely upon the values of a single tuple, whereas vertical operators apply to a set of tuples in the relation.

In the following sections, we will discuss the components of relational and domain algebra that are relevant to the Relix extensions we introduce in chapter four. The reader will also find some helpful examples for a better understanding of Relix.

3.3 Relational Algebra

Relational algebra is used to create relations, extract data from them, and make them interact with each other to produce more useful information. The following are the basic families of operations.

3.3.1 Assignment

Definition

Assignment is the most basic operation in Relix. It is used to create new relations or append data to an existing relation. There are two types of assignments, the first instantly interprets and executes the assignment, while the other just interprets the expression and stores it as a view in Relix's internal structures.

Example

The simplest forms of assignment is copying one relation to another:

- NEW_RELATION ← OLD_RELATION

In this case NEW_RELATION is immediately created. If we just need to create a view, however, we would use the "is" operator. The following expression

- FUTURE_RELATION is [MV_NAME, MV_YEAR] where (MV_DIRECTOR = "TERRY GILLIAM") in MOVIE;

is compiled and stored for future use.

Syntax

<identifier> ← <rel-expression>;

< identifier> is <rel-expression>;

<rel-expression> ≡ <rel-name> | <projection> <selection> <rel-expression>
| . <projection> <selection> <rel-expression>
| # <projection> <selection> <rel-expression>
| <join-expression>
| <scalar-expression>

3.3.2 Selection

Definition

A selection defines a new relation based on a subset of the *tuples* of a given relation. This is also referred to as a horizontal selection. The selection is based on the value of a logical expression called a *predicate* which is evaluated on every tuple of the relation.

Example

To retain only municipalities with a population larger than 500K, we can use the following statement:

➢ POPULATION_500 ← where (MU_POPULATION > 500000) in MUNICIPALITY;

Syntax

<selection> ≡ in | <T-Selector> | <QT-Selector>

The syntax for the <T-Selector> and the <QT-Selector> will be given in sections 3.3.4 and 3.3.5, respectively.

Note

Please note that the predicate is a single boolean value, evaluated on a single tuple in the relation. In nested queries, the clause after the where statement may include conditions on relations that would be considered nested within the relational expression after the in operator.

3.3.3 Projection

Definition

A projection defines a new relation based on a subset of the *attributes* of a given relation. This is also referred to as a vertical selection.

Example

In order to create a new relation NAMES which contains only the names of the municipalities in the MUNICIPALITY relation, we project the relation MUNICIPALITY on the attribute MU_NAME, as follows:

➢ NAMES ← [MU_NAME] in MUNICIPALITY;

Syntax

<projection> ≡ <empty> | [<domain-list>]

<domain-list> ≡ <domain-list>, <domain-name> | <domain-name>

Note

In flat relational databases, the projection list is composed uniquely of atomic attributes. In the case of nested queries, we will expand the definition of a projection to include relational expressions as well.

3.3.4 T-Expressions

Description

In the original Relix documentation, T-Expressions are viewed as the combination of a projection (The horizontal bar of the T) and a T-Selection (The vertical bar of the T). This is the simplest form of relational expressions.

Example

Assuming we needed the names of municipalities with a population larger than 500K, then we would formulate the query using the following T-Expression:

- MU_NAME_POP_500 ← [MU_NAME] where (MU_POPULATION > 500000) in MUNICIPALITY;

[MU_NAME] specifies a projection on the attribute MU_NAME, the condition (MU_POPULATION > 5000000) is the selection condition, and MUNICIPALITY is the name of the relation where the query is applied.

Syntax

<T-Selector> ≡ where <domain-expression> in

3.3.5 QT-Expressions

Description

QT-Expressions, or quantifiers, provide a means in Relix to count the number of different values of an attribute in a relation. They are based on two notations, namely the counter "#", and the proportion symbol, "." (a dot). The first counts the number of tuples satisfying a given condition, and the second computes the proportion of those tuples with respect to a larger set of tuples.

QT-Expressions can be divided into three groups, namely QT-Counters, QT-Predicates and QT-Selectors. QT-Counters and QT-Predicates return scalar values, either numeric or boolean. QT-Selectors return relations based on a count or proportion criteria. The best way to understand QT-Expressions is using examples.

Example

We will illustrate QT-Expressions using the movies relation described in chapter two. To calculate the number of movies in 1995 (assuming each movie has a unique name), we use the following QT-Counter expression:

- MOVIE_1995 ← # MV_NAME where (MV_YEAR =1995) in MOVIE;

It evaluates to 2.

If we need the proportion of movies with a duration longer than 100 minutes, then we use the proportion notation as follows:

➢ PROP_LONGER_100 ← . MV_NAME where (MV_DURATION > 100) in MOVIE;

and that evaluates to 0.6, which is the number of movies longer than 100 minutes (3) divided by the total number of movies (5).

Finally, if we need the list of directors with at least two movies in 1995, we could formulate that query using the following QT-Selector:

➢ TWO_95 ← [MV_DIRECTOR] where { (# ≥ 2) MV_NAME }, (MV_YEAR = 1995) in MOVIE;

Syntax

<QT-Counter> ≡ . <projection> <selection> <rel-expression> |
<projection> <selection> <rel-expression>

<QT-Selector> ≡ where { <quantifier-list> } in |
where { <quantifier-list> }, <domain-expression> in

<quantifier-list> ≡ <QT-Predicate> | <quantifier-list> , <quantifier-list>

<QT-Predicate> ≡ (<domain-expression>) <domain-expression>

Note

In the above examples, the count operator is applied to atomic attributes. But in the context of nested queries, an attribute can be atomic or composed. That raises a question not only about the syntax of QT-Counters applied to nested relations, but also the semantics of such expressions. What does the expression

➢ # [SHOPPING_MALL] in MUNICIPALITY;

mean? Should it return a single or multiple tuples? How is it translated into flat relational algebra? For now, we will essentially look at QT-Expression as inner-

most expressions in a nested query, leaving the discussion on expressions similar to the above to chapter seven.

3.3.6 μ-Joins

Description

The μ-joins - read mu-joins - are a family of binary operators on relations. They are defined as follows:

- ijoin ≡ natjoin: Intersection join, also called natural join;
- ujoin: Union join;
- djoin ≡ dljoin: Difference join, also called difference left join;
- sjoin: Symmetric difference join;
- ljoin: Left join;
- rjoin: Right join;
- drjoin: Difference right join;

To help the reader's intuition, we will show through examples the use of each of the joins.

Example

A typical example demonstrating the μ-joins is that of a store's inventory. We define as sample data set the following two relations:

SALES (SALES_ID, ITEM_ID, QUANTITY_SOLD)

SALES_ID	ITEM_ID	QUANTITY_SOLD
AT5467	TS6832	4
AT5467	SK0970	3
AT5467	SH0098	2

Table 3-1 Tuples of the SALES relation

and

PURCHASE (PURCHASE_ID, ITEM_ID, QUANTITY_IN)

PURCHASE_ID	ITEM_ID	QUANTITY_IN
MJ0986	TS6832	20
LK9573	SK0970	15
LK9573	TD0943	8

Table 3-2 Sample data for the PURCHASE relation

The ijoin between the two relations gives the transactions for items that were both sold and bought in the sample data. The query could be expressed as follows:

IJOIN_PURCHASE_SALES ← [ITEM_ID, QUANTITY_IN, QUANTITY_SOLD] in PURCHASE ijoin SALES;

The result would be the following table:

ITEM_ID	QUANTITY_IN	QUANTITY_SOLD
TS6832	20	4
SK0970	15	3

Table 3-3 Result of the IJOIN_SALES_PURCHASE query

In order to demonstrate the union join, we will assume that the manager of the store wishes to put in a single relation the number of items sold and purchased for each item. The query could be expressed as follows:

- UJOIN_SALES_PURCHASE ← [ITEM_ID, QUANTITY_IN, QUANTITY_SOLD] in PURCHASE ujoin SALES;

The result of that query for the two tables given in this example is:

ITEM_ID	QUANTITY_IN	QUANTITY_SOLD
TS6832	20	4
SK0970	15	3
TD0943	8	DC
SH0098	DC	2

Table 3-4 Result of the UJOIN_SALES_PURCHASE query

where DC is a symbol for NULL values.

Similarly, the difference join between PURCHASE and SALES, expressed in the query

- DJOIN_PURCHASE_SALES ← [ITEM_ID, QUANTITY_IN, QUANTITY_SOLD] in PURCHASE djoin SALES;

results in all items that were purchased but never sold

ITEM_ID	QUANTITY_IN	QUANTITY_SOLD
TD0943	8	DC

Table 3-5 Results of the DJOIN_PURCHASE_SALES query

To help the reader visualize the μ-joins, we define in Figure 3-1 below the set B to be the intersection join between the PURCHASE and SALES relations, A to be the difference join and A ∪ B ∪ C the union join. The remaining μ-joins could then be described as follows:

- sjoin = A ∪ C;
- ljoin = A ∪ B;
- rjoin = B ∪ C;
- drjoin = C;

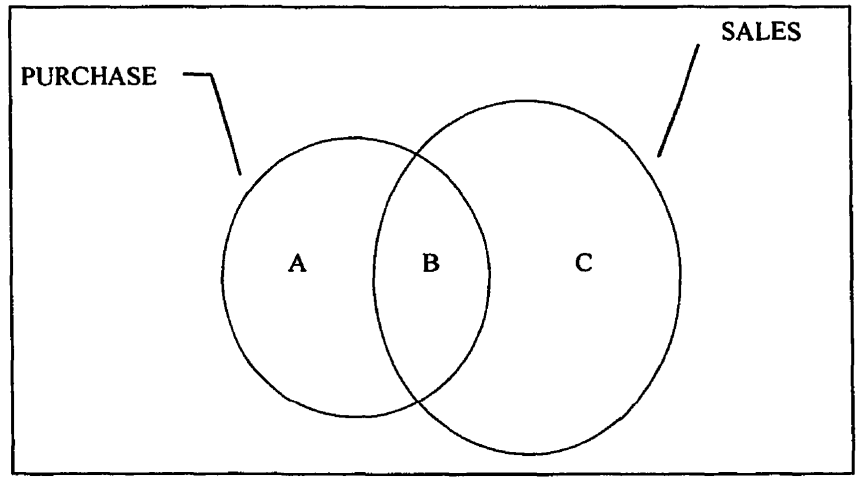


Figure 3-1 Analogy of μ-joins with set graphs

Finally, in the examples given above, the joins were applied on the common attribute between the two relations, namely ITEM_ID. Relix, however, allows joins on attributes with different names. Given relations R1 (X,Y) and R2 (Z,W), we can join R1 and R2 on attributes Y and Z as follows:

➢ R3 ← R1 [Y ijoin Z] R2;

Syntax

<identifier> ← <join-expression>;

<join-expression> ≡ <rel-expression> <join-mode> <rel-expression>

<join-mode> ≡ <μ-join> | [<domain-list> <μ-join> <domain-list>]

<μ-join> ≡ijoin | natjoin | ujoin | djoin | dljoin | sjoin | ljoin | rjoin | drjoin

3.3.7 σ-joins

The sigma_joins played a minor role in this research, and hence due to scope limitations, they will not be presented in this chapter.

3.4 Scalar Expressions

Description

In Relix, relations defined on a single attribute can appear in scalar expressions. For example, given two relations P (A) and Q (B), with A and B real integers, the following expression is legal in Relix:

- R ← (P + Q) / 2;

Relix produces the Cartesian product of P and Q, computes the expression

(A + B) / 2

for every tuple in the Cartesian product, and then assigns the result to R. Since R has no defined attributes from the assignment, Relix automatically generates a system attribute name. The name selected depends on the domain of (P + Q) / 2.

Example

To demonstrate Relix's behavior with scalar expressions, we will assume that the relations P (A) and Q (B) have the following sample data sets:

A
1
2
3

Table 3-6 Sample data for the relation P

B
4
5
6

Table 3-7 Sample data for the relation Q

The expression

- $R \leftarrow (P + Q);$

evaluates to

.real
5
6
7
8
9

Table 3-8 Result of R ← (P + Q)

and the expression

➢ R ← (Q – P);

evaluates to

.real
1
2
3
4
5

Table 3-9 Result of R ← (Q – P)

Syntax

<scalar-expression> ≡ 'unary-relation' | (<scalar-expression>) |
<unary-op> <scalar-expression> |
<function-op> (<scalar-expression>) |
<scalar-expression> <binary-op> <scalar-expression>

<unary-op> ≡ '+' ; '–' ; 'not' ; '!' ; '~' ; 'eval'

<function> ≡ 'abs' ; 'acos' ; 'asin' ; 'atan' ; 'ceil' ; 'chr' ; 'cos' ; 'cosh' ; 'floor' ; 'isknown' ; 'log' ; 'log10' ; 'ord' ; 'round' ; 'sin' ; 'sinh' ; 'tan' ; 'tanh'

<binary-op> ≡ '+' ; '–' ; '*' ; '/' ; 'and' ; '&' ; 'or' ; '|' ; '<' ; '<=' ; '=' ; '~=' ; '!=' ; '>=' ; '>' ; 'max' ; 'min' ; 'mod' ; 'pow' ; 'cat' ; '||'

3.5 Empty Projections

Description

In Relix, it is also possible to project on an empty set of attributes. Relix creates a system generated attribute called .bool, and the result is stored in a relation with a single boolean value (true or false).

Example

The following example returns the value true if at least one municipality in the MUNICIPALITY relation has a population larger than 500K, and false otherwise.

- BOOLEAN_500 ← [] where (MU_POPULATION > 500000) in MUNICIPALITY;

For the data set given in section 2.3, the result of this query would be true, as shown in Table 3-10.

.bool
true

Table 3-10 Result of the empty projection BOOLEAN_500

This possibility in Relix will prove to be very useful in transforming atomic predicates into relational predicates in section 4.2.1.

3.6 Domain Algebra

Domain algebra has been the main focus of our research, since in nested queries we widen the notion of domain to include relational attributes as well as atomic attributes. So a good understanding of the nature of domain algebra is essential to the understanding of the extensions we introduce to Relix.

Let us assume that a teacher has set up a relation to store the grades of his students. It could look like

GRADES (STUDENT_ID, EXAM_1, EXAM_2, FINAL)

with the following tuples:

STUDENT_ID	EXAM_1	EXAM_2	FINAL
9110769	78	83	85
9234598	89	82	87
9289765	85	79	80
9124659	87	92	90

Table 3-11 Tuples of the relation GRADES

At the end of the semester, the teacher needs to calculate the final grade for each student, the class average, and the class distribution. The final grade is calculated as follows:

20% * EXAM_1 + 30% * EXAM_2 + 50% * FINAL

In order to represent this formula, we need to create a virtual domain called FINAL_GRADE which would be applied to the relation GRADES.

- let FINAL_GRADE be (0.2*EXAM_1 + 0.3*EXAM_2 +0.5*FINAL);

Then we need to instantiate the virtual attribute by associating it with a relation, in this case GRADES, and assigning the final result to a new relation called MARKS:

- MARKS ← [STUDENT_ID, FINAL_GRADE] in GRADES;

STUDENT_ID	FINAL_GRADE
9110769	83
9234598	85.9
9289765	80.7
9124659	90

Table 3-12 The MARKS relation

Please notice that the formula for the final grade depends uniquely on attributes of the same tuple. This type of operators are called horizontal operators, as opposed to vertical operators the result of which depends on the attributes of several tuples, and sometimes all the tuples of the relation.

In order to calculate the class average, for example, the teacher needs a vertical operator that would sum the grades and then divide by the number of the students. Relix offers several vertical operators, the simplest of all being the red operator, which stands for reduction.

Using the red operator, this is how we may compute the class average:

- let SUM be red + of FINAL_GRADE;
- let STUDENT_NUM be red + of 1;
- let AVERAGE be SUM / STUDENT_NUM;
- CLASSE_AVERAGE ← [AVERAGE] in MARKS;

AVERAGE
84.9

Table 3-13 The CLASS_AVERAGE

That was a brief introduction to domain algebra. In the next two sections, we will examine with more details the horizontal and the vertical operators in Relix.

3.6.1 Horizontal Operators

Description

As explained above, horizontal operators apply to attributes of the same tuple. They are classified in three categories, namely unary operators, functions and binary operators.

Unary operators, including { +, – , not ...}, are the first set of horizontal operators. Along with functions, { abs, cos, sin ...}, they apply to a single attribute, and are of no particular interest to our study.

Binary operators are grouped into commutative/associative operators and order operators. An operator ω is commutative if $\omega(a,b) = \omega(b,a)$ and associative if $\omega(a,\omega(b,c)) = \omega(\omega(a,b), c)$. For instance, $\max(A,B) = \max(B,A)$, whereas $(A-B) \neq (B-A)$.

Finally, the successor (succ) and predecessor (pred) operators, although unary, also belong to the order operators.

Syntax

<domain_expression> ≡ <domain-name> | (<domain-expression>) |
<unary-op> <domain-expression> |
<function> (<domain-expression>) |
<domain-expression> <binary-op> <domain-expression>

<unary-op> ≡ '+' ; '–' ; 'not' ; '!' ; '~' ; 'eval'; 'vir'

<function> ≡ 'abs' ; 'acos' ; 'asin' ; 'atan' ; 'ceil' ; 'chr' ; 'cos' ; 'cosh' ; 'floor' ; 'isknown' ; 'log' ; 'log10' ; 'ord' ; 'round' ; 'sin' ; 'sinh' ; 'tan' ; 'tanh'

<binary-op> ≡ '+' ; '–' ; '*' ; '/' ; 'and' ; '&' ; 'or' ; ' | ' ; '<' ; '<=' ; '=' ; '~=' ; '!=' ; '>=' ; '>' ; 'max' ; 'min' ; 'mod' ; 'pow' ; 'cat' ; '||'

<associative-op> ≡ '+' ; '*' ; 'and' ; '&' ; 'max' ; 'min' ; 'or' ; '|'

<order-op> ≡ '+' ; '–' ; '*' ; '/' ; 'and' ; '&' ; 'or' ; ' | ' ; 'max' ; 'min' ; 'mod' ; 'pow' ; 'cat' ; '||' ; 'pred' ; 'succ'

3.6.2 Vertical Operators

Description

Reduction and functional mapping are the two forms of vertical operators. The first produces a single result and applies to a single attribute in the relation. The general form of reduction is

red <associative-op> of <domain-expression>

The result is obtained by cumulatively applying the associative operator to the domain expression of all tuples. The order in which the operator is applied is irrelevant to the operation; hence the operator should be associative and commutative.

Equivalence reduction is similar to simple reduction, but tuples are grouped according to a given attribute. The syntax for equivalence reduction is:

equiv <associative-op > of <domain-expression > by <domain-list>

The other form of vertical operators is functional mapping. It differs from reduction in that functional mapping gives a different result for every tuple in the relation. In fact, the result at any given tuple level depends on the tuple itself as well as all the tuples before it. Therefore, functional mapping depends on the order induced by one or more attributes. The syntax for functional mapping is given by

par <order-op> of <domain-expression> order <domain-list> by <domain-list>

Example

In order to illustrate the concept of equivalence reduction, let us go back to the GRADES relation and modify it slightly. Assuming the professor teaches three courses, and wants to put all grades in the same relation, the relation becomes

ALL_GRADES (COURSE, STUDENT_ID, EXAM_1, EXAM_2, FINAL)

with the following tuples:

COURSE	STUDENT_ID	EXAM_1	EXAM_2	FINAL
CS318-612	9110769	78	83	85
CS318-612	9234598	89	82	87
CS318-575	9289765	85	79	80
CS318-575	9124659	87	92	90

Table 3-14 Tuples of the relation ALL_GRADES

Computing the average of each student remains the same as before, but we instantiate the domain expression onto ALL_GRADES rather than GRADES:

- ALL_MARKS ← [COURSE, STUDENT_ID, FINAL_GRADE] in ALL_GRADES;

The average per course would be calculated as follows:

- let ALL_SUM be equiv + of FINAL_GRADE by COURSE;
- let ALL_STUDENT_NUM be equiv + of 1 by COURSE;
- let ALL_AVERAGE be ALL_SUM / ALL_STUDENT_NUM;
- ALL_CLASSE_AVERAGE ← [COURSE, ALL_AVERAGE] in ALL_MARKS;

The results of both queries are shown in the tables below:

COURSE	STUDENT_ID	FINAL_GRADE
CS318-612	9110769	83
CS318-612	9234598	85.9
CS318-575	9289765	80.7
CS318-575	9124659	90

Table 3-15 Tuples of the relation ALL_MARKS

COURSE	ALL_AVERAGE
CS318-612	84.45
CS318-575	85.35

Table 3-16 Tuples of the relation ALL_CLASS_AVERAGE

Example

In order to illustrate the use of functional mapping, we will assume that the professor in the above example has kept records of his earnings for the past five years, and needs to find out how much money was accumulated at the end of each year. The professor stored his data in a relation called EARNINGS, which has the following form:

EARNINGS (SCHOOL, YEAR, SALARY)

with the following data set:

SCHOOL	YEAR	SALARY
Concordia	1991	60000
Concordia	1992	65000
McGill	1992	65000
McGill	1994	78000
McGill	1995	80000

Table 3-17 The professor's earnings in the last five years

In order to compute the cumulative earnings in the past five years, the professor needs to define a virtual attribute as follows:

- let CUM_SALARY be fun + of SALARY order YEAR;

and then instantiate it on EARNINGS using the expression

- TOTAL_EARNINGS ← [YEAR, CUM_SALARY] in EARNINGS;

to give the following result:

YEAR	CUM_SALARY
1991	60000
1992	125000
1994	203000
1995	283000

Table 3-18 Result of TATAL_EARNINGS

In the same manner we introduced grouping to reduction, when tuples are grouped by an attribute in the relation, functional mapping (fun) becomes partial functional mapping (par).

Syntax

<domain-expression> ≡ <vertical-op>

<vertical-op> ≡ red <associative-op> of <domain-expression> |
equiv <associative-op> of <domain-expression> by <domain-list> |
fun <order-op> of <domain-expression> order <domain-list> |
par <order-op> of <domain-expression> order <domain-list>
by <domain-list> |
par <order-op> of <domain-expression> by <domain-list>
order <domain-list>

3.7 Closure

Definition

Relix has the closure property, so any selection on one or more relations results in a relation. Although that property allows the user to successively apply selections to the result of previous queries, except for QT-Counters, that result can never be used in a scalar expression, even when it is a singleton.

Example

In section 3.6, we computed the class average for the sample data in the MARKS relation, and stored the result in the relation

AVERAGE
84.9

Table 3-19 The CLASS_AVERAGE relation

Even though the content of the relation CLASS_AVERAGE is a single value, we cannot use it as a scalar. So the expression

➢ PASSING_STUDENTS ← [STUDENT_ID] where (FINAL_GRADE > CLASS_AVERAGE) in MARKS;

is illegal in Relix. We will find out in chapter four that this property is sometimes undesired and will learn how it can be countered.

3.8 Order of Execution

Description

Before we plunge into a full discussion of the proposed extensions to Relix, we would like to expose Relix's order of execution. In any given relational expression, the following rules apply:

1. Joins have precedence over T-Expressions and QT-Expressions;
2. All joins have the same precedence;
3. Join expressions are executed from left to right;
4. T-Expressions and QT-Expressions are executed from right to Left;
5. Parentheses can change the default order of execution;

Example

Relix interprets the query

➢ MY_STUDENTS ← [STUDENT_ID, AVERAGE] where (COURSE = "CS318-612") in ALL_GRADES ijoin ALL_MARKS;

in the following order

1. Executes the expression (ALL_GRADES ijoin ALL_MARKS);

2. Selects tuples satisfying (COURSE = "CS318-612");

3. Projects on the attributes STUDENT_ID and AVERAGE;

4. Calls the result MY_STUDENTS;

4. User's Manual

4.1 Introduction

This chapter aims at introducing the extensions proposed for Relix. It serves as a user's manual for someone who wants to build nested queries in Relix. The user is not required to be familiar with the low level data modeling in order to understand this chapter. Throughout chapter four, objects are conceived as mutually nested, regardless of the way nesting is implemented. In chapter five, we will link nested queries to the implementation of the data model and show how nested queries may be transformed into a series of flat relational expressions.

4.2 Changes Proposed to Relix

We recall that the main difference between flat relations and nested relations is that in the context of nested relations, an attribute can be atomic or composed (relational). Hence the scope of the domain algebra widens naturally to accommodate relational expressions where only domain expressions were allowed with flat relations.

In the following sections, we examine the two basic components of relational algebra, namely selection and projection, in the context of nested relations. For each component, we discuss the meaning of domain expressions when we substitute relational domains for atomic domains. In the third section, we add another feature to Relix, namely *attribute migration*, which allows the user to port an attribute from one relation to another via the link between the two relations. Each section of this chapter is divided into three parts, a description of the proposed modification, one or more examples supporting the description and

explaining how the new facility can be used in a query, and finally the syntax of the new Relix statement.

4.2.1 Selection

Description

In flat relational algebra, a selection clause resolves to a boolean expression. A selection condition is evaluated on each tuple in the relation, and only those tuples that satisfy that condition are selected.

In queries involving complex objects, the selection condition may be tested on the child relation as well as the parent relation. We call this condition an *inner* or *embedded* selection, and express it using the syntax for empty brackets described in section 3.5 of chapter three. For example, the general form of an embedded T-Selection is given by the following:

> where ([] where «inner-selection» in «inner-rel-expression») in «outer-rel-expression»

The inner selection clause may be of any selection form allowed in Relix. The interpretation of that selection, however, will not be the same as that of the flat selection. In fact, chapter five is dedicated to examples with different types of inner selections and how they are translated into flat relational and domain algebra.

Example

We would like to demonstrate the use of inner selections with the following example. Let us assume that we are asked to name all municipalities containing large shopping malls. The answer would require the SHOPPING_MALL relation to be nested within the MUNICIPALITY relation. Since the condition for

selecting the municipalities depends on the inner relation, we apply the extension proposed above.

In this case,

«inner-selection» = "where SM_AREA > 10000 in"

and

«inner-rel-expression» = "SHOPPING_MALL"

The nested query would be

➢ Q ← [MU_NAME] where ([] where SM_AREA > 10000 in SHOPPING_MALL) in MUNICIPALITY;

Syntax

<selection> ≡ in | T-Selector | QT-Selector |
where ([] <selection> <rel-expression>) in <rel-expression>

4.2.2 Projection

Description

The adjustments brought to the projection clause in a nested query can be stated in very simple terms. Instead of allowing only atomic domain names in the projection list, we include relation names as well. The foremost challenge of this evolution, however, is to maintain the symmetrical model after the projection.

At the conceptual level, we would like to think about every relation name in the projection list as a new object mutually nested with the parent entity in the query. However, defining the objects in the implementation depends on the nature of the relational expression in the projection. In order to prepare the reader for the

discussion in chapter five, this section describes the two types of relational expressions in a projection list, followed by two illustrating examples.

Some expressions represent only a view on an existing relation, and those are labeled *object preserving expressions*. Others, called *object generating expressions*, generate new objects that do not maintain a one-to-one relationship with the original entity from which they spanned. In Relix, the first type of expressions, also called *horizontal relational expressions*, satisfy one or more of the following conditions:

- The relational expression has no projection list;
  - ➢ Q ← where (MU_AREA > 50000) in MUNICIPALITY;
- The projection list of the relational expression contains at least one domain name from the projected relation;
  - ➢ Q ← [MU_NAME] in MUNICIPALITY;
- The projection list of the relational expression contains at least one horizontal domain expression;
  - ➢ let DENSITY be (MU_POPULATION / MU_AREA);
  - ➢ Q ← [DENSITY] in MUNICIPALITY;

The second type of expressions, also called *vertical relational expressions*, are defined as relational expressions where *all* domain expressions in the projection list are given in terms of vertical operators.

➢ Let MOVIES_PER_YEAR be equiv + of 1 by MV_YEAR;

➢ Q ← [MOVIES_PER_YEAR] in MOVIE;

Example

We define large shopping malls as

- SM_LARGE is [SM_NAME] where (SM_AREA > 10000) in SHOPPING_MALL;

Then, if we need to find large shopping malls in large municipalities, we include the relation name SM_LARGE in the projection list, along with the municipality name, as follows

- MU_SM_LARGE ← [MU_NAME, SM_LARGE] where (MU_POPULATION > 500000) in MUNICIPALITY;

The query would result in two new relations, MU_SM_LARGE and SM_LARGE, which form, along with the relationship between them, a new couple of mutually nested objects. It is important to note, nonetheless, that every tuple in SM_LARGE corresponds to a single shopping mall. Hence, SM_LARGE maintains a one-to-one relationship with the relation from which it spanned.

Example

We assume that in some comparative study on the tourist attractions in municipalities, we are asked to provide a list of municipalities along with the number of restaurants in each municipality. Taking advantage of nested expressions, we express that query as follows:

- let RS_NB be red + of 1;
- RS_COUNT is [RS_NB] in RESTAURANTS;
- ANSWER ← [MU_NAME, RS_COUNT] in MUNICIPALITY;

The details of how such expressions are handled is given in section 5.4.2. For now, we just need the result of RS_COUNT in order to pursue our argument. In flat relational algebra, RS_COUNT would be a singleton relation with a single tuple holding the total number of restaurants in the relation RESTAURANTS.

In the context of nested queries, RS_COUNT would be a unary relation, where each tuple corresponds to the number of restaurants per municipality. In other words, every tuple in RS_COUNT corresponds to a municipality and not to a restaurant. So the relationship between RS_COUNT and the restaurants is broken. That is a major difference with the first example, and will prove to be crucial at the implementation level in chapter five.

Syntax

<projection> ≡ <projection> , <rel-name> | <projection>, <domain-name> |
<rel-name> | <domain-name>

Note

Despite the consistency in the conceptual data representation, we will find out in chapter five that implementation is slightly different depending on the nature of the relation in the projection list.

4.2.3 Attribute Migration

Description

We recall from section 3.7 that attributes in one relation cannot be used as scalars in a T-Selector involving the parent relation. Researchers in the domain of nested relations have introduced the NEST and UNNEST operations [3, 47] to resolve that problem (among others). We argued in chapter one, however, that our definition of nested queries does not require those facilities. As an alternative, we introduce the notion of attribute migration which allows us to bring an attribute

from one relation to another, without actually changing the structure of both relations.

Example

Imagine having two relations,

EMPLOYEE (EM_ID, EM_NAME, EM_AGE, EM_DEPARTMENT)

and

CHILDREN (CH_ID, CH_NAME, CH_AGE)

linked by

EMCH_LINK (EM_ID, CH_ID)

We are asked to compute the average age difference between a parent and a child for each department. One way to implement the query is to join the three relations together and then apply flat domain algebra on EM_AGE, CH_AGE and EM_DEPARTMENT.

A more interesting way is to use nested queries. If we do so, we create the following nested view of the two relations:

EMPLOYEE (EM_NAME, EM_AGE, EM_DEPARTMENT, CHILDREN)
(CH_NAME, CH_AGE)

At this point, we need to compute the difference between EM_AGE and CH_AGE. But those two attributes belong to different relations, and Relix does not allow us to have domain expressions involving attributes from different relations. In addition, any relational operation on CHILDREN, by the closure property, will itself result in a relation. So the way to solve this problem is to bring CH_AGE up one level from CHILDREN to EMPLOYEE. We call that facility *attribute migration.*

In the implementation of attribute migration, we propose using scalar expressions. We recall from chapter three that any relation defined on a single attribute can be used both as a scalar and a relation. Similarly, a nested singleton relation can be transformed into an atomic attribute of the parent relation. The following sequence of Relix expressions demonstrates the concept:

- CHILD_AGE is [CH_AGE] in CHILDREN;
- NEW_EMPLOYEE is [EM_DEPARTMENT, EM_AGE, CHILD_AGE] in EMPLOYEE;
- let AGE_DIFF be EM_AGE – CHILD_AGE;
- let SUM_AGE_DIF be equiv + of AGE_DIFF by EM_DEPARTMENT;
- let COUNT_AGE_DIF be equiv + of 1 by EM_DEPARTMENT;
- let AVG_AGE_DIFF be SUM_AGE_DIFF/COUNT_AGE_DIFF;
- A ← [EM_DEPARTMENT, AVG_AGE_DIFF] in NEW_EMPLOYEE;

Please note how CHILD_AGE was defined as a relation using the "is" assignment operator, and was later used as a scalar in computing the age difference AGE_DIFF. Most important, the user does not have to specify explicitly the link between EM_AGE and CHILD_AGE. By including CHILD_AGE as an attribute of NEW_EMPLOYEE, it became part of the same complex object which has EM_AGE as attribute. In section 5.5.1, we will show how a nested query involving attribute migration is translated into a series of flat relational expressions.

4.3 Is That All?

At first look, the proposed extensions to Relix seem minimal, and possibly insufficient. However, the simplicity and the invariance of our model (always one link relation for every couple of related objects) saves us from dealing with restructuring operators such as NEST and UNNEST as in [3, 4, 46, 47] . In addition, the symmetry in our model makes all queries equivalent in treatment and performance, so we need not duplicate any information nor create redundant objects. Finally, we did not make any distinction between object preserving operators and object creating operators, so we did not have to introduce special operators such as the **Project** operator in [20].

We believe, therefore, that the modifications discussed in this chapter are sufficient to cover all potential queries on mutually nested relations. In fact, we will present in the next chapter over fifteen queries of all types, expressing them in nested format as well as the equivalent flat format, and providing the reader with a full description of the translation from one syntax to the other.

5. Implementation

5.1 Introduction

Having defined the new syntax proposed to Relix, we now proceed to working out an algorithm for a parser which transforms nested queries into a series of flat queries. We choose to work this problem bottom up, starting with a series of examples that cover most aspects of nested queries, and leading to a formal algorithm in section 5.6. All the queries are based on the sample data proposed in chapter two, hoping the reader would find it useful to follow the evolution of the queries on the same data set.

The queries of this chapter are grouped into four sections, namely *Flat Queries*, *Nested Expressions in the Selection Clause*, *Nested Expressions in the Projection Clause* and *Miscellaneous queries*. Each of the queries includes a description of the query, an English statement of the query, the nested query, the equivalent query expressed in flat Relix and finally a discussion about the major issues concerning that query.

5.2 Flat Queries

Description

This example shows how flat Relix queries can still be applied without the parser even knowing anything about nesting.

Query

Name all municipalities with population larger than 500K.

Nested expression

- Q ← [MU_NAME] where (MU_POPULATION > 500000) in MUNICIPALITY;

Translated code

None.

Discussion

The parser detects no nesting, and the query is executed as a flat query.

5.3 Nested Expressions in the Selection Clause

5.3.1 Horizontal Expression with no Predicates

Description

In this first example on nested queries, we test the existence of one object in another.

Query

Name all municipalities that have shopping malls in them.

Nested expression

- Q ← [MU_NAME] where ([] in SHOPPING_MALL) in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- S2 ← [MU_ID] in S1;

- S3 ← S2 ijoin MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S3;

Discussion

We see here how the inner relational expression replaces the boolean expression in the selection clause. The steps to follow in transforming the nested query into a series of flat queries are the following:

1. Join the inner relation (SHOPPING_MALL) with its link relation to the outer relation (MUSM_LINK);
2. Augment the empty projection by the identifier of the outer relation, namely MU_ID;
3. Join the result of 2 (S2) with the outer relation (MUNICIPALITY);
4. Project S3 on the projection clause of the original relational expression containing only domain names. The system uses MU_ID to generate the object identifier of the new relation.

5.3.2 Expression Showing the Symmetry in the Data Model

Description

This query is given in comparison with the query in section 5.3.1. It shows how a relation can be a parent or a child relation depending on the access mode to the information.

Query

Name all shopping malls in the municipality of Longueuil;

Nested expression

- Q ← [SM_NAME] where ([] where MU_NAME = "LONGUEUIL" in MUNICIPALITY) in SHOPPING_MALL;

Translated code

- S1 ← MUNICIPALITY ijoin MUSM_LINK;
- S2 ← [SM_ID] where MU_NAME = "LONGUEUIL" in S1;
- S3 ← S2 ijoin SHOPPING_MALL;
- Q ← [MU_ID, SM_NAME] in S3;

Discussion

While the previous query returns all the municipalities having a shopping mall, this query returns the shopping malls of a given municipality. So the MUNICIPALITY relation becomes the child relation and the SHOPPING_MALL relation becomes a parent relation. Both queries use the same link relation, namely MUSM_LINK, and require the same amount of effort to evaluate. Most important, no restructuring is needed when we switch from one query to another.

5.3.3 Horizontal Expression, a Single Predicate

Description

This query introduces predicates into the selection clause. Therefore, we need to apply a selection criterion on the inner relation before joining it with its parent relation.

Query

Name all municipalities with shopping malls larger than 10000m^2.

Nested expression

- Q ← [MU_NAME] where ([] where SM_AREA > 10000 in SHOPPING_MALL) in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- S2 ←[MU_ID] where SM_AREA > 10000 in S1;
- S3 ← S2 ijoin MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S3;

Discussion

1. Again, the first step is to join the inner relation with its link relation to the outer relation;
2. Next, we need to apply the selection criteria on the inner relational, namely SHOPPING_MALL. Since this relation is nested in the MUNICIPALITY relation, the selection will be made on S1.

   In addition, please notice how we include in the outcome of S2 the identifier of the outer relation. We will need that identifier when we join back S2 with MUNICIPALITY in S3.

   So the golden rule is that whenever we translate a nested expression of the form

   [] where <inner-selection> <inner-rel-name>

into a full Relix expression, we add to the projection list the identifier of the outer relation.

3. We join S2 with the outer relation to get all the needed attributes of MUNICIPALITY;

4. Project on MU_NAME and MU_ID to get the final result;

One might argue that steps 1 and 2 can be formulated more efficiently, namely as follows:

- S1 ← [SM_ID] where SM_AREA > 10000 is SHOPPING_MALL;

- S2 ← S1 ijoin MUSM_LINK;

Performing the selection on SHOPPING_MALL before the join with MUSM_LINK is with no doubt faster than the suggested sequence. In fact, the "*select then join*" sequence works well as long as we have horizontal expressions in the selection clause. Unfortunately, as will be shown in section 5.3.4, that it is not the case with vertical operators.

Since performance and optimization are not major issues in this work, we favored the simple algorithm to one which would have to read ahead, detect the type of selection and then back track and perform the query. What we could do, however, is replace S1 and S2 by a single statement, namely

- S1 ← [MU_ID] where SM_AREA > 10000 in SHOPPING_MALL ijoin MUSM_LINK;

For clarity, however, we decide to generate two assignments instead of one.

5.3.4 A Reduction Expression

Description

We mentioned previously that vertical expressions have a special treatment when they appear in a nested expression. We start with the simplest of vertical operators, the reduction statement.

Query

Name all municipalities where the total area of shopping malls exceeds 100000m^2.

Nested expression

- let TOT_AREA be red + of SM_AREA;
- Q ← [MU_NAME] where ([] where (TOT_AREA > 100000) in SHOPPING_MALL) in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- let TOT_AREA be equiv + of SM_AREA by MU_ID;
- S2 ← [MU_ID] where (TOT_AREA > 100000) in S1;
- S3 ← S2 ijoin MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S3;

Discussion

We solve this query by computing the total area of shopping malls in each municipality, and then selecting those with total area greater than 100000 m^2. The translated code can be explained as follows:

1. Join the inner relation with its link to the outer relation;

2. If we instantiate the domain expression

   let TOT_AREA be red + of SM_AREA;

   in the SHOPPING_MALL relation, that would result in the total area of all shopping malls. But we need the total shopping mall area per municipality. For that reason, the vertical expression appears as a nested expression in the query.

   As we recall from chapter three, the reduction operator does not allow grouping, but the equivalence reduction operator does. So, as a general rule, a reduction statement appearing in a nested expression is translated into an equivalence reduction, with grouping on the identifier of the outer relation.

   In the case of the example in hand, the outer relation is MUNICIPALITY, so the reduction is transformed into an equivalence reduction with grouping on MU_ID:

   let TOT_AREA be equiv + of SM_AREA by MU_ID;

3. Translate the nested expression into a full Relix expression by adding the identifier of the outer relation and assigning it to S2;

4. Join S2 with the MUNICIPALITY relation to get back all the MUNICIPALITY attributes;

5. Finally, project S3 on all the atomic attributes of the original expression, in this case MU_NAME, including MU_ID. The system then uses MU_ID to generate new unique identifiers.

5.3.5 A QT-Predicate

Description

In addition to horizontal and vertical expressions, the inner expressions in a nested query may contain QT-Expressions. With this example, we start the study of QT-Expressions by looking at QT-Predicates.

Query

Name all municipalities where more than three quarters of the shopping malls have areas less than 20000 m^2.

Nested expression

- Q ← [MU_NAME] where ([] where (. > 0.75) SM_NAME, SM_AREA < 20000 in SHOPPING_MALL) in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- S2 ← [MU_ID] where (. > 0.75) SM_NAME, SM_AREA < 20000 in S1;
- S3 ← S2 ijoin MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S3;

Discussion

It turns out that QT-Predicates need no special treatment.

1. The first step, as before, is to join the inner relation with the link relation.
2. The inner expression is then converted into a flat Relix expression by appending the identifiers to the empty brackets. That also tells Relix to group S1 by MU_ID before performing the proportion operator.
3. We join S2 with MUNICIPALITY to get the attributes we need for the projection;
4. We project S3 on MU_NAME and MU_ID to obtain the result required.

5.3.6 A QT-Counter

Description

The other type of QT-Expressions to examine are the QT-Counters in the nested expression. In fact, QT-Counters are allowed in inner expressions uniquely with a predicate, otherwise they make no sense.

Query

Name all municipalities that have more than ten shopping malls in them.

Nested expression

- Q ← [MU_NAME] where ((# SM_NAME in SHOPPING_MALL) > 10) in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;

- S2 ← [MU_ID] where ((# > 10) SM_NAME) in S1;
- S3 ← S2 ijoin MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S3;

Discussion

QT-Counters cannot be treated as such in nested relations. They have to be transformed into QT-Predicates before evaluation. That is the only way we would be able to group the counting by municipality. This translation is always possible, and the two expressions are equivalent.

In statement S2, we demonstrate how the translation may be done without loss or modification of the original information.

5.3.7 Two "anded" Predicates

Description

In all the examples we examined till now in chapter five, there was at most one predicate in the selection clause. But what happens if we have two predicates or more? How do we translate a nested query with multiple predicates into flat Relix expressions?

Query

Name all shopping malls in Longueuil that have Italian restaurants in them.

Nested Expression

- Q ← [SM_NAME] where (([] where MU_NAME = "LONGUEUIL" in MUNICIPALITY) and ([] where RS_TYPE = "ITALIAN" in RESTAURANT) in SHOPPING_MALL;

Translated code

- S1 ← RESTAURANT ijoin SMRS_LINK;
- S2 ← [SM_ID] where RS_TYPE = "ITALIAN" in S1;
- S3 ← MUNICIPALITY ijoin MUSM_LINK;
- S4 ← [SM_ID] where MU_NAME = "LONGUEUIL" in S3;
- S5 ← S4 ijoin S2;
- S6 ← S5 ijoin SHOPPING_MALL;
- Q ← [SM_ID, SM_NAME] in S6;

Discussion

This query involves not two but three relations, two of which are nested in the third. In addition, the selection involves two conditions, (MU_NAME = "LONGUEUIL") and (RS_TYPE = "ITALIAN").

We propose that each predicate be treated as if it was the only predicate in the inner selection. Then, depending on the logical operator between the two boolean expressions, we link the two results by a corresponding μ-join.

S1 and S2 represent the instantiation of the second predicate in the selection clause. Similarly, S3 and S4 represent the instantiation of the first predicate in the selection clause. The "and" operator in the inner expression is mapped onto an ijoin. If we recall from chapter three, the ijoin is the intersection of two relations; so if we apply it to S2 and S4, the result would be the tuples that satisfy both the first and the second conditions.

Once we have S5, the rest is similar to the previous examples: we join S5 with the SHOPPING_MALL relation to get all the attributes, then project on SM_NAME and SM_ID to obtain the final result.

But what happens if we had an operator different from the "and" operator? The answer is given in the following two sections.

5.3.8 Two "ored" Predicates

Description

In the previous example, we saw that an "and" operator between two predicates in an inner selection is mapped onto an ijoin in the translated Relix code. The reader might have guessed that in the same manner, an "or" operator translates into a ujoin.

Query

Name all shopping malls that have an Italian restaurant or an Odeon movie theater.

Nested expression

- Q ← [SM_NAME] where (([] where RS_TYPE = "ITALIAN" in RESTAURANT) or ([] where MT_COMPANY = "ODEON" in MOVIE_THEATRE)) in SHOPPING_MALL;

Translated code

- S1 ← MOVIE_THEATRE ijoin SMMT_LINK;
- S2 ← [SM_ID] where MT_COMPANY = "ODEON" in S1;
- S3 ← RESTAURANT ijoin SMRS_LINK;

- S4 ← [SM_ID] where RS_TYPE = "ITALIAN" in S3;
- S5 ← S4 ujoin S2;
- S6 ← S5 ijoin SHOPPING_MALL;
- Q ← [SM_ID, SM_NAME] in S6;

Discussion

The procedure for translating the nested query into flat expressions is identical to that of the previous example, except that S5 is the ujoin of S2 and S4 rather than the ijoin.

The case for a "not" operator is trickier than that of the "and" and the "or" operators. In conjunction with the "and" operator, the "not" operator maps well into the djoin relational operator. Yet, it has no immediate equivalent when it is placed alone in front of the predicate or in conjunction with an "or" operator. The following example demonstrates the procedure for handling negation.

5.3.9 The "not" Operator

Description

In our study, we handle the negation of a predicate by introducing a djoin into the translated code. The original expression in which the inner selection is applied is djoined with the result of the inner selection.

Query

Name all shopping malls that have no Italian restaurants.

Nested expression

- Q ← [SM_NAME] where (not ([] where RS_TYPE = "ITALIAN" in RESTAURANT)) in SHOPPING_MALL;

Translated code

- S1 ← RESTAURANT ijoin SMRS_LINK;
- S2 ← [SM_ID] where RS_TYPE = "ITALIAN" in S1;
- S3 ← S1 djoin S2;
- S4 ← S3 ijoin SHOPPING_MALL;
- Q ← [SM_ID, SM_NAME] in S4;

Discussion

From the example given above, we see that the "not" operator is translated onto a djoin of S1 with the translation of the inner expression. The result would be, as required, all the tuples in S1 that do not satisfy the stated condition.

Once we have computed S3, we would just need to join back with the parent relation to get all the attributes of SHOPPING_MALL, and then project the result on SM_NAME and SM_ID.

5.3.10 Four Predicates with Parentheses

Description

When dealing with more than two predicates in an inner expression, the parentheses can play an important role in determining the precedence of evaluation of each predicate. In nested queries, it is no different, as we will find out in the following example.

Query

Name all shopping malls in Longueuil that are smaller than 20000 m^2 or in Laval and are smaller than 10000 m^2.

Nested expression

- Q ← [SM_NAME] where ((([] where MU_NAME = "LONGUEUIL" in MUNICIPALITY) and (SM_AREA < 20000)) or (([] where MU_NAME = "LAVAL" in MUNICIPALITY) and (SM_AREA < 10000))) in SHOPPING_MALL;

Translated code

- S1 ← where SM_AREA < 10000 in SHOPPING_MALL;
- S2 ← MUNICIPALITY ijoin MUSM_LINK;
- S3 ← [SM_ID] where MU_NAME = "LAVAL" in S2;
- S4 ← S1 ijoin S3;
- S5 ← where SM_AREA < 20000 in SHOPPING_MALL;
- S6 ← MUNICIPALITY ijoin MUSM_LINK;
- S7 ← [SM_ID] where MU_NAME = "LONGUEUIL" in S6;
- S8 ← S5 ijoin S7;
- S9 ← S4 ujoin S8;
- S10 ← S9 ijoin SHOPPING_MALL;
- Q ← [SM_ID, SM_NAME] in S10;

Discussion

The lesson to draw from this example is that the parser respects the precedence order imposed by the parentheses. The translated code clearly shows how the two "and" operators are applied before the "or" operator.

5.4 Nested Expressions in the Projection Clause

5.4.1 Horizontal Relational Expression

Description

This is the first example of a query containing a relation name in the projection list. It is identical to the query of the first example in section 4.2.2.

Query

Name all municipalities that have a population over 500000 and the shopping malls in them that have an area greater than 10000 m^2.

Nested expression

- SM_LARGE is [SM_NAME] where SM_AREA > 10000 in SHOPPING_MALL;
- Q ← [MU_NAME, SM_LARGE] where MU_POPULATION > 500000 in MUNICIPALITY;

Translated code

- S1 ← where MU_POPULATION > 500000 in MUNICIPALITY;
- Q ← [MU_ID, MU_NAME] in S1;

- S2 ← SHOPPING_MALL ijoin MUSM_LINK;
- SM_LARGE ← [SM_ID, SM_NAME] where SM_AREA > 10000 in S2;
- <link-relation name> ← [SM_ID, MU_ID] in (where SM_AREA > 10000 in S2) ijoin S1;

Discussion

Unlike queries we have seen till now, this query has no nesting in the selection clause, but contains a relation name in the projection list. Handling projection lists starts after Relix encounters the closing bracket "]" of the projection clause - Remember that Relix parses T-Expressions from right to left. At that point, it has already computed the relation on which the domain list would be projected, namely S1.

Before dealing with non atomic attributes in the projection list, Relix creates a new relation Q, which is the projection of all atomic attributes in the domain list on S1. In the case of this example, there is only on such attribute, namely MU_NAME. The object identifier of MUNICIPALITY is included in the projection list of Q.

The relation SM_LARGE in the projection list was defined as a T-Selector view on the SHOPPING_MALL relation. Since it is nested in the projection list of the outer query on MUNICIPALITY, Relix proceeds by joining SHOPPING_MALL to its link relation with the parent relation. The result of the join is stored in S2.

As mentioned in chapter four, every relational expression in the projection list results in a new object mutually nested with the parent object in the query. That means that new object identifiers should be generated for the new objects. In the case of SM_LARGE, it was defined using only horizontal operators, and hence every tuple in it still has a one to one correspondence with tuples in

SHOPPING_MALL. Therefore, we include SM_ID in the projection list of SM_LARGE.

Finally, Relix creates a new link relation between Q and SM_LARGE. The new link relation in this case is a subset of MUSM_LINK and includes only the tuples where both MU_ID and SM_ID exist in Q and SM_LARGE, respectively. It can be generated more efficiently, however, using S1 and S2. Relix generates automatically a name for the link relation, then renames the MU_ID and SM_ID in Q, SM_LARGE and their link relation.

5.4.2 Vertical Relational Expression in the Projection Clause

Description

In the previous example, we saw how horizontal relational expressions are dealt with at the implementation level when they appear in the projection list of a nested query. This query, which was introduced as the second example of section 4.2.2, demonstrates the equivalent treatment for vertical relational expressions.

Query

Name municipalities along with the number of restaurants in each of them.

Nested Expression

- let RS_NB be red + of 1;
- RS_COUNT is [RS_NB] in RESTAURANTS;
- Q ← [MU_NAME, RS_COUNT] in MUNICIPALITY;

Translated code

- S1 ← RESTAURANTS ijoin MURS_LINK;
- Q ← [MU_ID, MU_NAME] in MUNICIPALITY;
- let RS_NB be equiv + of 1 by MU_ID;
- RS_COUNT ← [MU_ID, RS_NB] in S1;
- let RS_COUNT_ID be MU_ID;
- let Q_ID be MU_ID;
- <link-relation name> ← [RS_COUNT_ID, Q_ID] in RS_COUNT ijoin Q_ID;

Discussion

At the conceptual level, each relational expression appearing in the projection list of a nested query gives birth to a new set of objects. When the relational expression in the projection list is a horizontal expression, each new object corresponds to a tuple from the original relation, which was the case in the previous example. In this query, if we include the restaurant identifier in the projection list of RS_COUNT, we would obtain for the geographical data of chapter two the following result:

RS_ID	RS_NB
RS1	3
RS2	3
RS3	1
RS4	2
RS5	3
RS6	2

Table 5-1 RS_COUNT with RS_ID

The number of restaurants (RS_NB) for RS1, RS2 and RS5 all represent the same entity, namely the number of restaurants for municipality MU1. Similarly, RS4 and RS6 fall in the same municipality. In our opinion, grouping the result of a vertical relational expression by the identifier of the child relation in a nested query leads to redundant objects. A better solution would be to group RS_COUNT by the municipality OIDs, leading to the following result for RS_COUNT:

MU_ID	RS_NB
MU1	3
MU2	2
MU3	1

Table 5-2 RS_COUNT with MU_ID

This result is more compact and more accurate. Therefore, horizontal relational expressions are grouped by the object identifier of the child relation in the query, whereas vertical relational expressions are grouped by the object identifier of the parent relation in the query.

Again, Relix generates a link relation between RS_COUNT and Q. The link relation in this case is not a subset of MURS_LINK. It is constructed initially as the join of RS_COUNT and Q by renaming MU_ID in each relation, which would look as follows:

RS_COUNT_ID	Q_ID
MU1	MU1
MU2	MU2
MU3	MU3

Table 5-3 Initial link relation between RS_COUNT and Q

Then, when Relix reassigns unique object identifiers for each tuple in Q and RS_COUNT, it reassigns new OIDs to the link relation. Q_ID follows the renumbering of the Q relation, and RS_COUNT_ID follows that of RS_COUNT. The resulting link relation may look as follows:

RS_COUNT_ID	Q_ID
RC1	Q1
RC2	Q2
RC3	Q3

Table 5-4 Final link relation between RS_COUNT and Q

5.4.3 Another Type of Horizontal Relational Expressions

Description

To end this section on relational expressions in projection lists, we provide the reader with an example that helps us make a clear distinction between horizontal and vertical relational expressions.

Query

Name the largest and the smallest shopping malls in every municipality

Nested Expression

- let MAX_AREA be red max of SM_AREA;
- let MIN_AREA be red min of SM_AREA;
- MAX_MIN_SM is [SM_NAME] where (MAX_AREA = SM_AREA) or (MIN_AREA = SM_AREA) in SHOPPING_MALL;
- Q ← [MU_NAME, MAX_MIN_SM] in MUNICIPALITY;

Translated code

- Q ← [MU_ID, MU_NAME] in MUNICIPALITY;
- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- let MAX_AREA be equiv max of SM_AREA by MU_ID;
- let MIN_AREA be equiv min of SM_AREA by MU_ID;
- MAX_MIN_SM ← [SM_ID, SM_NAME] where (MAX_AREA = SM_AREA) or (MIN_AREA = SM_AREA) in S1;
- <link-relation name> ← [MU_ID, SM_ID] in (where (MAX_AREA = SM_AREA) or (MIN_AREA = SM_AREA) in S1) ijoin MUNICIPALITY;

Discussion

At first look, this example might look more complex than the previous ones. In fact, it just contains more expressions in the original nested query. The treatment is no different from what we have encountered till now.

First, the atomic attributes in the projection list are projected on MUNICIPALITY. Then, the Relix parser encounters a relation name in the projection list and attempts to evaluate it as a nested expression. It therefore joins the child relation with the link relation.

Next, every domain expression involving a vertical operator is converted into a vertical expression with grouping by the parent relation identifier. So

- let MAX_AREA be red max of SM_AREA;

becomes

- let MAX_AREA be equiv max of SM_AREA by MU_ID;

and

- let MIN_AREA be red min of SM_AREA;

becomes

- let MIN_AREA be equiv min of SM_AREA by MU_ID;

At that point, MAX_MIN_SM is evaluated on S1 rather than the SHOPPING_MALL relation. Since MAX_MIN_SM contains a domain name in its projection list, it is considered as a horizontal relational expression, even if its selection clause includes vertical domain expressions. Therefore, SM_ID and not MU_ID, is appended to the projection list of MAX_MIN_SM. The final result is composed of MAX_MIN_SM, Q and the link relation as constructed in section 5.4.1.

5.5 Miscellaneous Queries

5.5.1 Abstraction From a Relation to an Attribute

Description

We explained in chapter four how atomic attributes in a relation can be extracted from that relation and migrated into another relation. In the context of a nested query, that concept is very useful if we need to bring an attribute from a child relation into a parent relation in a given query.

Query

Get the ratio of area covered by restaurants in a shopping mall to the area of the shopping mall.

Nested expression

- let TOT_AREA be red + of RS_AREA;
- TOT_AREA_RS is [TOT_AREA] in RESTAURANT;
- SM_RS_AREA is [SM_NAME, SM_AREA,TOT_AREA_RS] in SHOPPING_MALL;
- let RATIO be (TOT_AREA_RS * 100) /SM_AREA;
- Q ← [SM_NAME, RATIO] in SM_RS_AREA;

Translated code

- SM_RS_AREA is [SM_ID, SM_NAME, SM_AREA] in SHOPPING_MALL;

- S1 ← RESTAURANT ijoin SMRS_LINK;
- let TOT_AREA be equiv + of RS_AREA by SM_ID;
- TOT_AREA_RS is [SM_ID, TOT_AREA] in S1;
- let RATIO be (TOT_AREA * 100) / SM_AREA;
- S2 ← SM_RS_AREA ijoin TOT_AREA_RS;
- Q ← [SM_ID, SM_NAME, RATIO] in S2;

Discussion

The translation of the query starts at the assignment which creates the relation Q. First, SM_RS_AREA is defined by projecting SHOPPING_MALL on all atomic attributes in the projection list of SM_RS_AREA. The second step is to translate TOT_AREA_RS. Since TOT_AREA_RS appears in the projection list of a relational expression involving the SHOPPING_MALL relation, we proceed by forming the ijoin of RESTAURANT with its link relation to SHOPPING_MALL.

The reduction in the domain expression which defines TOT_AREA is then translated into an equivalence reduction with grouping on SM_ID, as we are now computing the area of restaurants per shopping mall. Since TOT_AREA_RS is a vertical relational expression in the projection clause of SM_RS_AREA, its projection list is augmented by SM_ID, the object identifier for SM_RS_AREA.

Both SM_RS_AREA and TOT_AREA_RS are defined using the "is" operator, so they only define views on existing relations, and hence do not constitute new objects; therefore, their unique object identifiers don't need to be recalculated at this point in the translated code. The domain expression which expresses RATIO in terms of TOT_AREA_RS and SM_AREA is translated as is. TOT_AREA_RS

is a unary relation, so according to section 4.2.3, it may be used in a domain expression.

The last step in the translation is the projection of RATIO and SM_NAME on SM_RS_AREA. At the conceptual level, SM_RS_AREA and TOT_AREA_RS are mutually nested, so depending on the query, any of the two relations can be fully nested within the other. In the case of the query in hand, TOT_AREA_RS is embedded in SM_RS_AREA, so TOT_AREA_RS is perceived as a relational attribute of SM_RS_AREA. Knowing that RATIO involves an arithmetic operation between attributes from SM_RS_AREA and TOT_AREA_RS, we need to join those two relations.

This operation is allowed if and only if one of the two following cases applies:

1. Both relations are view definitions and have the same object identifier;
2. Both relations are full assignments and represent new objects in the database.

In the first case, both relations are joined together on the common attribute without the help of the link relation. In the second case, each relation would have a different object identifier, and hence the two relations may only be joined via the link relation.

In this example, the first condition is satisfied, hence we join SM_RS_AREA and TOT_AREA_RS directly, and then project the result on SM_ID, SM_NAME and RATIO.

5.5.2 Two Levels of Nesting

Description

We end the series of examples with a query involving two levels of nesting. Expressions involving more nesting levels are also possible, and they are treated no different than the two level nesting.

Query

Name all shopping malls with more than ten restaurants, along with their corresponding municipality.

Nested expression

- let TOT_RS be red + of 1;
- NEW_RS is [RS_NAME, TOT_RS] in RESTAURANT;
- SM_10RS is [SM_NAME] where ([] where TOT_RS > 10 in NEW_RS) in SHOPPING_MALL;
- Q ← [MU_NAME, SM_10RS] in MUNICIPALITY;

Translated code

- S1 ← SHOPPING_MALL ijoin MUSM_LINK;
- S2 ← RESTAURANT ijoin SMRS_LINK ijoin MUSM_LINK;
- let TOT_RS be equiv + of 1 by MU_ID, SM_ID;
- NEW_RS is [MU_ID, SM_ID, RS_ID, RS_NAME, TOT_RS] in S2;
- S3 ← [MU_ID, SM_ID, RS_ID] where (TOT_RS > 10) in NEW_RS;

- S4 ← S3 ijoin SHOPPING_MALL;
- SM_10RS ← [SM_ID, SM_NAME] in S4;
- Q ← [MU_ID, MU_NAME] in MUNICIPALITY;

Discussion

The query can be divided into two subqueries, finding shopping malls that have more than ten restaurants in them, and then finding municipalities that contain such shopping malls.

If we proceed to evaluate the query from right to left, we encounter the first level of nesting between the SHOPPING_MALL relation and the MUNICIPALITY relation. So we evaluate the first link between SHOPPING_MALL and its link relation to MUNICIPALITY, namely MUSM_LINK.

Next, the parser encounters NEW_RS which is defined on the RESTAURANT relation. At this point, the parser knows that this is the second level of nesting, and hence, instead of linking the RESTAURANT relation to SMRS_LINK, it links it to (SMRS_LINK ijoin MUSM_LINK). That way, we have in S2 the unique identifiers of both the SHOPPING_MALL and the MUNICIPALITY relations.

From that point on, the unique identifier for the parent relation of RESTAURANT is (MU_ID, SM_ID). This difference is clearly noticed in the domain algebra expression: the reduction is converted to an equivalence with grouping on both MU_ID and SM_ID. Similarly, we append to the empty brackets in S3 both MU_ID and SM_ID.

The rest of the translation is trivial. We join with SHOPPING_MALL to reintroduce the shopping mall attributes in S4, and then project on SM_NAME and SM_ID (SM_10RS is a horizontal relational expression). Additionally, the

atomic attributes are projected on MUNICIPALITY and assigned to Q. Relix then creates new object identifiers for Q and SM_10RS and generates a link relation between the two new entities.

5.6 An Algorithm for Parsing Nested Queries

Throughout the queries of chapter five, we have provided the reader with a translation of each nested query into a sequence of flat relational expressions. Time has come to describe the algorithm for a parser which receives at the input a nested query and generates at the output the translated code for the current implementation of Relix. We would like to remind the reader that we have not implemented this algorithm in our work, and that it covers only the syntax covered in the chapters four and five of the thesis.

The algorithm is called upon whenever the Relix parser detects a nested expression:

1. DETECTION

   The nested query flag is raised when the parser encounters a relational expression in place of a domain expression. In the selection clause, that corresponds to a relational expression instead of a boolean expression, and to a relational expression instead of an attribute name in the projection list.

2. RELATIONAL EXPRESSION IN THE SELECTION CLAUSE

2.1. Every predicate in the selection clause is evaluated. If it involves a nested relation, then we call the procedure EVALUATE_NESTED, otherwise we call the procedure EVALUATE_FLAT.

2.2. If a not operator is found in front of an expression, the relation on which the expression is evaluated is djoined with the outcome of step 2.1.

2.3. Next, the parser concatenates the generated relations. For every boolean operator between predicates corresponds a join operation, as shown in Table 5-5. The relations resulting from step 2.2 are hence joined, in the same order the boolean expressions would have been computed (that is respecting the order imposed by parentheses). From the concatenation results a single relation, containing the unique identifier of the parent.

2.4. Join the relation of 2.3 back with the parent relation.

2.5. We project the atomic domain names on the relation of the previous step, including the OID of the outer relation. The result is a single relation R.

Logical expression	Relational expression
and	ijoin
or	ujoin
not , !	djoin

Table 5-5 Correspondence table between the logical operators and the μ-join

3. RELATIONAL EXPRESSION IN THE PROJECTION LIST

3.1. For every relational expression in the projection list, we evaluate this relation by calling the EVALUATE_NESTED procedure. For every such projection P_i, we get a new relation C_i. The object identifier is selected as the object identifier of the parent entity in the query if C_i is a vertical relational expression, and as the object identifier of the child entity in the query if C_i is a horizontal relational expression.

3.2. For every pair (R, C_i), we create a new link relation, based on the following rules:

3.3. If C_i is a vertical relational expression, then the new link relation is the projection of the object identifiers of R and C_i on the union join between R and C_i;

3.4. If C_i a horizontal relational expression, the new link relation is the projection of the object identifiers of R and C_i on the union join between R, C_i, and the original link relation between the ancestors of R and C_i.

The EVALUATE_NESTED and EVALUATE_FLAT functions may be described as follows:

EVALUATE_NESTED

The evaluation of a nested relational predicate takes place exactly as a flat relation, with the following modifications:

- Instead of applying the query to the relation after the in operator, we apply it to the ijoin of the nested relation with its link relation to the parent(s) relation(s);
- We augment the empty projection list by the identifier of the outer relation(s);
- If we encounter a QT-Predicate, the expression is translated literally, as it is expressed in the nested query. However, if the relational expression contains a QT-Counter in a predicate, it is translated into a QT-Predicate before evaluation.
- The

  red <associ_op> of <dom_expr> ;

  translates into

  equiv <associ_op> of <dom_expr> by parent_id(s);

- The

  equiv <associ_op> of <dom_expr> by <dom_list>

  translates into

  equiv <associ_op> of <dom_expr> by <dom_list>, parent_id(s);

- The

  fun <order_op> of <dom_expr> order <dom_list>;

  translates into

  par <order_op> of <dom_expr> order <dom_list> by parent_id(s);

- The

  par <order_op> of <dom_expr> order <dom_list> by <dom_list

  translates into

  par <order_op> of <dom_expr> order <dom_list> by <dom_list>, parent_id(s);

- The

  par <order_op> of <dom_expr> by <dom_list> order <dom_list>;

  translates into

  par <order_op> of <dom_expr> by <dom_list>, parent_id(s) order <dom_list>;

EVALUATE_FLAT

When this function is called, it means that the predicate is not a relational expression. It is just a boolean condition on the parent relation in the query. The predicate is therefore translated into the following relational expression:

<system_name> is [<parent_id>] where <condition> in <parent_relation>;

6. Nested SQL Queries

6.1 Introduction

In recent years, SQL has become accepted as the industry standard language for relational database access. It is widely implemented and supported by large commercial database management systems such as ORACLE, SYBASE and DB2.

Many attempts have been made in the past to extend SQL and make it operational on nested relations [50, 7]. What we describe in this chapter is not as complete as some of the literature, and serves only as a proof of concept.

We will start by giving a general presentation of SQL, highlighting only those topics that will be covered by the examples on nested queries. Then, we will analyze the potential changes required for the general SQL statement to fit within the mutually nested objects model. Finally, we will give some examples to illustrate the proposed extensions to SQL.

6.2 General Presentation of SQL

Many books and manuals have been written about SQL since SEQUEL2, the original SQL language, was described by Chamberlin et al. in [51]. Even the American National Standard Institute (ANSI) and the International Standards Organization (ISO) have adopted SQL as the standard language for relational database management systems. The reader can find literature on ANSI SQL in [52] and on ISO SQL in [53].

One benefit of SQL is that the same language provides commands for a wide range of tasks such as

- querying data;
- creating and modifying objects;
- inserting deleting and updating tuples in tables;

Following the same pattern used with Relix, we will focus only on the SQL query facilities, at the heart of which is the SELECT statement.

The basic SELECT statement in SQL has the following structure:

```
SELECT <domain-expression>
FROM   <relational-expression>
WHERE <condition>
```

In comparison with Relix, the <domain expression> is equivalent to the projection list, the <relational-expression> clause is equivalent to the relational expression after the "in" operator, and the <condition> is equivalent to the selection clause. If we need, for example, to find the names of movies directed by "Steven Speilberg", we would express the SQL query as follows:

```
SELECT mv_name
FROM   movie
WHERE mv_director = 'STEVEN SPEILBERG';
```

The extension from SQL to nested SQL is discussed next. We will first propose a syntax for expressing nested queries, then show through examples how nested SQL can be translated back into standard SQL.

6.3 Changes Proposed to SQL for Handling Nested Queries

The difference between flat and nested queries is that only atomic attributes are allowed in flat queries, whereas nested queries can have atomic as well as

composed attributes. Hence, in porting SQL to nested SQL, we need to allow relations where only atomic attributes were previously allowed. In the simple SELECT statement we introduced in the previous section, the changes would affect the WHERE clause - discussed in section 6.3.1- as well as the SELECT clause - discussed in section 6.3.2. In section 6.3.3, we compare single row functions with group functions in SQL, and introduce the special treatment required for group functions.

6.3.1 Selection: The WHERE Clause

SQL provides the equivalent of the "[] where" clause we introduced in chapter four, namely the EXISTS operator. The expression

EXISTS <subquery>

evaluates to TRUE if the SQL subquery returns at least one row.

For instance, if we need to select the municipalities that have large shopping malls, we express the query in the following standard SQL statement:

```
SELECT  mu_name
FROM    municipality, musm_link
WHERE   EXISTS
        (SELECT *
        FROM    shopping_mall, musm_link
        WHERE   sm_area > 10000 AND
                municipality.mu_id = musm_link.mu_id AND
                shopping_mall.sm_id = musm_link.sm_id)
```

The above expression, however, is very complex, and references the object identifiers explicitly. So we would like to reduce it simply to

```
SELECT  mu_name
FROM    municipality
WHERE   EXISTS
        (SELECT *
        FROM    shopping_mall
        WHERE   sm_area > 10000)
```

This way, we hide from the user all references to the link relations and the unique identifiers, without altering the existing SQL syntax.

6.3.2 Projection: The SELECT Clause

Since attributes may be relational expressions, we will allow SELECT statements into the projection list of another SELECT statement. So the expression

```
SELECT  mu_name, (SELECT   sm_name
                  FROM     shopping_malls
                  WHERE    sm_area > 10000)
FROM    municipality
WHERE   mu_population > 500000
```

will be a legal statement. We have not introduced new keywords to standard SQL, but we have modified the syntax of the projection clause in the SELECT statement to allow nested selection. The whole concept of nested SELECT statements in the projection list of other SQL queries is still absent from standard SQL. It has been introduced, however, into other extensions to SQL, such as SQL/NF [7], although that was in the context of one-directional nesting.

6.3.3 Functions in SQL

Before we illustrate the use of the proposed extensions to SQL in answering queries, we need to discuss SQL functions, as they play an important role in nested queries. As we may recall from chapter three, we identified in Relix two types of expressions, vertical and horizontal. And we showed how vertical operators need a special treatment when used in the context of nested queries.

Well, SQL is no different. SQL functions can be divided into single row functions and group functions. Single row functions, such as ABS(), SIN() and TRIM(), are equivalent to horizontal expressions, for their result depends on a single tuple. SUM, COUNT, MIN and the rest of the group functions return results based on a group of rows, just as the vertical operators in Relix.

Whenever a group function appears in a nested selection, we need to group the nested relation by the parent identifier before we execute the function. The following expression is an illustration of the concept:

```
SELECT  mu_name
FROM    municipality
WHERE   (SELECT COUNT(*)
         FROM    shopping_mall) > 3;
```

The COUNT() function in the inner SELECT statement counts the number of shopping malls. Since it is embedded in an outer select statement involving the MUNICIPALITY relation, COUNT() would logically have to return the number of shopping malls per municipality. The same query could not be expressed in standard SQL without using the GROUP BY clause and making direct references to the object identifiers. Integrating the notion of nesting within SQL reduces the complexity of the expression and releases the user from worrying about tuple references.

6.4 Examples of Nested Queries

In this section, we choose three nested queries among those discussed in chapter five and show how they can be expressed using nested SQL.

6.4.1 SELECT Clause in the WHERE Condition

Query 5.3.3

Name all municipalities with shopping malls larger than 10000m^2.

Nested expression

```
SELECT  mu_name
FROM    municipality
WHERE   EXISTS
        (SELECT  *
        FROM     shopping_mall
        WHERE    sm_area > 10000);
```

Translated code

```
SELECT  mu_id, mu_name
FROM    municipality
INTERSECT
SELECT  mu_id
FROM    shopping_mall, musm_link
WHERE   (sm_area > 10000) AND
        (shopping_mall.sm_id = musm_link.sm_id)
```

Discussion

The first thing to do in the translation of the nested query is to evaluate the inner condition, which can be decomposed into three components. The EXISTS operator, combined with the SELECT * expression, are replaced by a selection on the object identifier of the parent relation in the query, namely MU_ID. The inner query on SHOPPING_MALL is translated into a query on SHOPPING_MALL and its link relation to MUNICIPALITY, namely MUSM_LINK. Finally, in addition to the original condition on the shopping mall area (sm_area > 10000), the selected tuples should satisfy the additional condition *(shopping_mall.sm_id = musm_link.sm_id)*, which guaranties that they have an entry in MUSM_LINK.

Once the inner relation is resolved, we form the outer relation. The translation involves extending the selection clause to include the object identifier of the MUNICIPALITY relation in the statement. Finally, we build the intersection of the two relations to get the final result consisting of tuples satisfying both the outer and the inner conditions.

6.4.2 SELECT Clause in the Projection List

Query 5.4.1

Name all municipalities that have a population over 500000 and the shopping malls in them that have an area greater than 10000 m^2.

Nested expression

```
SELECT mu_name, (SELECT sm_name
                 FROM   shopping_mall
                 WHERE sm_area > 10000)
FROM   municipality
WHERE  mu_population > 500000;
```

Translated code

```
SELECT  mu_id, mu_name
FROM    municipality
WHERE   (mu_population > 500000);
```

```
SELECT  sm_name, sm_id
FROM    shopping_mall, musm_link
WHERE   (sm_area > 10000) AND
        (shopping_mall.sm_id = musm_link.sm_id)
```

Discussion

The treatment of relational expressions in the projection list of a nested SQL query is identical to that realized in Relix. At the conceptual level, the result is composed of two new mutually nested relations. At the implementation level, new object identifiers are created for the new objects, and a new link relation establishes the relationship between the two objects. The nested query is therefore translated into two distinct queries.

The first SELECT statement stands for the outer selection in the nested query. The projection list is however extended to include the object identifier of the municipality. The translation of the embedded SELECT statement is less trivial, and the changes are listed below:

1. We have included the shopping mall identifier in the projection list. As we recall from chapter four, horizontal relational expressions preserve the objects of the relation in which they are instantiated, and hence the identifiers of those objects can be reused. If the nested SELECT statement was a vertical relational expression, the OID of the parent object in the

query would have been included in the projection list of the translated code.

2. The selection is made in SHOPPING_MALL and MUSM_LINK, since we need to select only those shopping malls that are linked to municipalities in the database. That criterion is met thanks to the added condition in the WHERE clause.

3. Although not shown above in the translated code, the nested SQL database manager would also create the link relation between the two new objects, and rename the object identifiers in all three relations.

6.4.3 Group Functions in the WHERE Clause

Query 5.3.4

Name all municipalities where the total area of shopping malls exceeds 100000m^2.

Nested expression

```
SELECT  mu_name
FROM    municipality
WHERE   (SELECT SUM(sm_area)
FROM    shopping_mall) > 100000;
```

Translated code

```
SELECT  mu_id, mu_name
FROM    municipality
INTERSECT
SELECT  mu_id
```

```
FROM    shopping_mall, musm_link
WHERE   (SELECT SUM(sm_area)
         FROM    shopping_mall, musm_link
         WHERE   (shopping_mall.sm_id = musm_link.sm_id)
         GROUP BY   mu_id) > 100000;
```

Discussion

The translation of this query is identical to that in section 6.4.1, except for the added GROUP BY clause. We argued in section 6.3.3 that whenever a group function appears in a nested selection, we need to group the nested relation by the parent identifier before we execute the function. The SUM() function is a group function, and it is used in the selection clause of a nested SELECT statement. Therefore, when we translate the nested query into a series of flat SQL statements, a GROUP BY clause is introduced. In the case of this example, the grouping is done on the identifier of the parent object, namely mu_id.

6.5 Comparison Between Nested and Standard SQL

If we compare the code for the nested query with the translated standard SQL statements, we see that the nested query enjoys several important properties over standard SQL. At the usage level, nested SQL expresses more naturally the statement of the query, which makes it more user friendly. It is almost a direct translation from the definition of the query.

Second, nested SQL expresses the query independently from the implementation model, so the user is not aware of how relationships between entities are represented in the system. In addition, it allows the same query to be ported transparently from one data model to another without modifications or adaptations. Even optimization would be done on the implementation side rather than on the query side.

In the standard SQL statement, the user has to reference the object identifiers explicitly. This is undesirable since OIDs are supposed to be hidden to the user and managed entirely by the system. The alternative would be to use foreign keys for referencing, a concept we argued against in chapter two.

The nested query is very simple and elegant compared to its translation. It holds no references to any of the object identifiers nor the link relations, and grouping is implied by the mere fact that one SELECT statement is embedded in another. In standard SQL, on the other hand, an explicit GROUP BY clause is mandatory to create groupings. In short, when it comes to expressing queries on mutually nested objects, nested SQL expressions are far more advantageous than standard SQL statements.

7. Conclusion and Future Work

We have presented in this thesis a new data model for objects which are mutually nested. Our data representation is characterized by symmetry, which enables us to achieve our main objectives, namely dynamic complex object definition and modeling of graph connected (rather than hierarchical) data structures. We believe that this system architecture is better suited for applications with multiple access patterns than traditional non-first-normal-form data models.

In addition, we introduced to the relational database system an important concept from the object-oriented paradigm, namely object identifiers. It helped us create direct and multiple object referencing in an efficient and consistent manner, without data or structure duplication. This modification has brought our model closer to the object model, without giving up the benefits of relational databases. In recent years, several authors such as Schek [20] and Stonebraker [21] have followed a similar approach. By designing data models that combine features from relational as well as object-oriented databases, they have laid out the foundations of a new "wave" of database management systems, the object-relational model.

In order to support our data model, we have proposed an extension to the Relix syntax which would enable users to deal with nested queries at a conceptual level. We offered a high level discussion of the necessary changes to the syntax of Relix, as well as a detailed explanation of how nested queries may be translated into flat relational algebra based on our data model. By examining over fifteen queries, ranging in difficulty from flat to doubly nested queries, we hope to have found an interesting way to get the reader familiar with all the problems involved in resolving nested queries. Finally, we proposed certain modifications to SQL in order to transform it into nested SQL, a language capable of dealing with mutually nested objects in an implicit way.

The domain of complex objects, however, is extremely vast. Just covering the literature on various complex object models and algebra, for example, is worth months of work, especially in that our model is rooted in both the relational and the object-oriented schools. There are various possibilities and options worth investigating which we decided to drop altogether because of scope and time limitations.

At the Relix level, for example, we have left out several interesting issues, such as the σ-joins. Let us consider for instance the following query,

- SM_LIST ← where (SM_AREA > 2000) in SHOPPING_MALL;

- Q ← [MU_NAME] where (SHOPPING_MALL ⊃ SM_LIST) in MUNICIPALITY;

where the ⊃ notation stands for the greater-than-join, and returns true if all tuples in SM_LIST are included in SHOPPING_MALL. We have not discussed how such expressions may be translated into flat relational algebra.

Another aspect missing from our discussion of chapters four and five is the effect of QT-Counters when applied to embedded relations in a given query. For example, what does the expression

- # [SHOPPING_MALL] in MUNICIPALITY

mean? The QT-Counter usually counts the number of different attribute values in a relation. But in this case, the attribute is itself a relation, so we need to do relational comparison which again calls upon σ-joins.

A third possibility for nested expressions involves μ-joins and the domain algebra. We saw in chapter three that a reduction expression takes the form

- let <domain-name> be red <associative-op> of <domain-expression>;

as in

- let SUM be red + of SM_AREA;

The only condition placed on the operator that follows the reduction is that it should be associative and commutative. Certain μ-joins, such as the ijoin for example, are both commutative and associative. So expressions such as

- let NEW_RELATION be red ijoin of MOVIE;

which are not allowed currently in Relix, should be legal in the context of nested queries. Moreover, when instantiated in the assignment

- Q ← [NEW_RELATION] in MOVIE_THEATER;

the result should return movies that are shown in all movie theaters. Unfortunately, the algorithm we presented in section 5.6 does not take into account that kind of statements. Fortunately, however, those queries are exceptions to the more common standard queries we discussed in chapter five. In fact, if any additional effort should be spent to complete our work, it should focus on implementing the data model and the Relix extensions we proposed, optimizing the query translations, and investigating new requirements for updates and integrity constraints in the context of mutually nested relations.

A.1 Relational Algebra

<rel-algebra> ::= <rel-expr> | <rel-expr> ← <rel-expr>
| <rel-expr> <+ <rel-expr>
| <rel-expr> [<dom-list> ← <dom-list>] <rel-expr>
| <rel-expr> [<dom-list> <+ <dom-list>] <rel-xpr>
| relation <rel-name> (<dom-list>)
| relation <rel-name> (<dom-list>) <rel-input>
| <rel-update>

<dom-list> ::= <dom-name> | <dom-list> , <dom-name>

<rel-input> ::= ← "unix-path" | ← { <tuple-list> }

<tuple-list> ::= (<const-list>) | <tuple-list> , (<const-list>)

<const-list> ::= <empty> | <constant> | <const-list> , <constant>

<constant> ::= <constant-in> : <dom-type> | <constant-in>

<constant-in> ::= dc | dk | true | false | NUMBER | "STRING"

EXAMPLES:

- domain A: short;
- domain B: short;
- domain C: string;

- relation R(A, B) ← "my-file";
- relation S(A, C) ← {(1, "one"), (2, dc), (3, "three")};
- t ← [A] in R;
- t [A <+ B] [B] in S;
- T initial R is T ujoin (T [B icom A] R);
- t ← T;

A.2 Relational Expressions

<rel-expr> ::= <rel-name> | <scalar-expr>
| (<rel-expr>) | pick <rel-expr>
| . <projection> <selection> <rel-expr>
| # <projection> <selection> <rel-expr>
| <projection> <selection> <rel-expr>
| <projection> vedit <rel-option>
| <projection> zorder <rel-expr>
| <rel-expr> <join_mode> <rel-expr>
| <rel-expr> { <const-list> }

<projection> ::= <empty> | [<dom-list>]

<selection> ::= in | where <dom-expr> in | where { <quan-list> } in
| where { <quan-list> } <dom-expr> in
| when <dom-expr> in | when <dom-expr> from

<rel-option> ::= <empty> | <rel-expr>

<join-mode> ::= <join> | [<dom-list> <join> <dom-list>]

<join> ::= nop | <mu-join> | <not> <sigma-join>

<not> ::= <empty> | ~ | ! | not

<mu-join> ::= ijoin | natjoin | djoin | ujoin | sjoin | ljoin | rjoin | dljoin | drjoin

<sigma-join> ::= gtjoin | gejoin | eqjoin | lejoin | ltjoin | div | icomp | sep | sub | sup

<quan-list> ::= (<dom-expr>) <dom-expr> | <quant-list>, <quan-list>

<dom-list> ::= <dom-list> , <dom-name> | <dom-name>

<const-list> ::= <const-list> , <constant> | <constant>

<constant> ::= <constant-in> : <dom-type> | <constant-in>

<constant-in> ::= dc | dk | true | false | NUMBER | "STRING"

EXAMPLES:

- domain A: short;
- domain B: short;
- domain C: string;
- domain D: boolean;
- relation R(A, C);
- relation S(B, D);

- t ← where A > 2*356 & C = "X" in R;
- t ← [B] where not D in S;
- t ← R ijoin R;
- t ← R [A ijoin B] S;
- t ← [A] where {(# = 2) C} in R;
- t ← [B] where {(. = 1/2) D} where D or B = 2 in R;
- t ←#[B] where {(. = 0.5) D} where B != 2 in R;
- t ←A { 3 };
- t ←vedit R;
- t ←[A, B] vedit;

A.3 Domain Algebra

<dom-algebra> ::= domain <dom-name> <dom-type>
| let <dom-name> be <dom-expr>
| let <dom-name> initial <dom-expr> be <dom-expr>

<dom-type> ::= any | bool | boolean | long | integer | intg
| real | float | string | strg | expr | expression
| stmt | statement | short | text

<dom-expr> ::= <dom-name>
| (<dom-expr>) | (<dom-type> <dom-expr>)
| <constant> | # | .
| <unary-op> <dom-expr> | <function> (<dom-expr>)
| <dom-expr> <binary-op> <dom-expr>
| if <dom-expr> then <dom-expr> else <dom-expr>
| <vertical-op>
| pick <scalar-name> | pick(<rel-expr>)

<constant> ::= <constant-in> : <dom-type> | <constant-in>

<constant-in> ::= dc | dk | true | false | NUMBER | "STRING"

<unary-op> ::= + | - | not | ! | ~ | eval | vir

<function> ::= abs | acos | asin | atan | ceil | chr | cos | cosh | floor |
isknown | log | log10 | ord | round | sin | sinh | tan | tanh

<binary-op> ::= + | - | * | / | and | & | or | | | < | <=
| = | ~= | != | >= | > | max | min | mod | pow | cat | ||

<associ-op> ::= + | * | max | min | and | & | or | |

<order-op> ::= + | - | * | / | and | & | or | | | cat | ||
| max | min | mod | pow | pred | succ | ~ | !~

<dom-list> ::= <dom-list> , <dom-name> | <dom-name>

<vertical-op> ::= red <associ-op> of <dom-expr>

| equiv <associ-op> of <dom-expr> by <dom-list>

| fun <order-op> of <dom-expr> order <dom-list>

| par <order-op> of <dom-expr> order <dom-list>
by <dom-list>

| par <order-op> of <dom-expr> by <dom-list>
order <dom-list>

EXAMPLES:

- domain A be long;
- domain B be integer;
- domain C be real;
- let X be (A + B) / 2;
- let Y be red * of A;
- let Z be equiv + of 1 by C;
- let T be fun + of pred B order A;

A.4 Scalar Expression

<scalar-expr> ::= <rel-set> | <rel-sequencial>

| (<scalar-expr>)

| (<type> <scalar-expr>)

| <unary-op> <scalar-expr>

| <fun-op> (<scalar-expr>)

| <scalar-expr> <binary-op> <scalar-expr>

| if <scalar-expr> then <scalar-expr> else <scalar-expr>

<type> ::= any | bool | boolean | long | integer | intg
| real | float | expr | expression | stmt | statement
| string | strg | short

<unary-op> ::= + | - | not | ! | ~ | eval

<function> ::= abs | acos | asin | atan | ceil | chr* | cos | cosh | floor |
isknown |
log | log10 | ord* | round | sin | sinh | tan | tanh

<binary-op> ::= + | - | * | / | and | & | or | | | < | <= | = | ~= | != | >= | >
| max | min | mod | pow | cat | ||

EXAMPLES:

- domain A real;
- relation R(A);
- relation S(A);
- t ← (R + R) / S;
- t ← round(R * 2);

References

[1] E. Codd. A Relational Model for Large Shared Data Bank, *ACM Communications*, June 1970, pp 377-387

[2] A. Makinouchi. A Consideration on Normal Form of Not-Necessarily-Normalised Relation in the Relational Data Model. *Proceedings of the International Conference on Very Large Databases*, pp 447-453, Tokyo, 1977.

[3] G. Jaeschke, H. J. Schek. Remarks on the Algebra of Non First Normal Form Relations, *Proceedings ACM SIGACT/SIGMOD Symposium on Principles of Database Systems*, Los Angeles, March 1982, pp 124-138.

[4] S. J. Thomas, P. C. Fisher. Nested Relational Structures. *Advances in Computing Research, The Theory of Databases*, P. C. Kanellakis, ed. JAI Press, Vol. 3, 1986, pp 269-307.

[5] K. Takeda. On the Uniqueness of Nested Relations. *Nested Relations and Complex Objects in Databases*, LNCS #361, eds. A. Abiteboul, P. C. Fischer, H.-J. Schek, Springer-Verlag, Heidelberg, 1989, pp. 139-150.

[6] M. A. Roth, H. F. Korth, A. Silberschatz. Theory of Non-First-Normal-Form Relational Databases. *Technical Report TR-84-36 (Revised January 1986)*, University of Texas at Austin, 1984.

[7] M. A. Roth, H. F Korth, D.S. Batory. SQL/NF: A Query Language for ¬1NF Relational Databases. *Information Systems* Vol. 12, No. 1, pp. 99-114, 1987.

[8] M. A. Roth, H. F. Korth, A. Silberschatz. Extended Algebra and Calculus for Nested Relational Databases. *ACM Transactions on Database Systems*, Vol. 13, No. 4, December 1988, pp 389-417.

[9] H.-J. Schek and P. Pistor. Data Structures for an Integrated Database Management and Information Retrieval System. *Proceedings on Very Large Databases*, Mexico City, Mexico, 1982.

[10] H.-J. Schek and M. H. Scholl. The Relational Model With Relation-Valued Attributes. *Information Systems*, Vol. 11, No. 2, 1986, pp 137-147.

[11] P. Pistor, F. Andersen. Designing a Generalised NF^2 Model with an SQL-Type Language Interface. *Proceedings of the Twelveth Conference on Very Large Databases*, Kyoto, Japan, 1986, pp. 278-288.

[12] M. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, A. Verroust. VERSO: A Database Machine Based on Nested Relations. *Nested Relations and Complex Objects in Databases*, LNCS #361, eds. A. Abiteboul, P. C. Fischer, H.-J. Schek, Springer-Verlag, Heidelberg, 1989, pp. 27-49.

[13] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, U. Deppisch. Architecture and Implementation of the Darmstadt Database Kernel System. *Proceedings ACM SIGMOD Conference on Management of Data*, San Francisco, 1987, pp 196-207.

[14] P. P. Chen. The Entity-relationship Model: Towards a Unified View of Data. *ACM TODS*, Vol. 1, No, 1, 1976.

[15] D. Shipman. The Functional Model and the Data Language DAPLEX. *ACM Transactions on Database Systems*, Vol. 6, No. 1, 1981, pp 140-173.

[16] M. Hammer and D. McLeod. Database Description with SDM: a Semantic Database Model. *ACM Transactions on Database Systems*, Vol. 6, No. 3, 1981, pp 351-386.

[17] E. Codd. Extending the Database Relational Model to Capture More Meaning. *ACM Transactions on Database Systems*, Vol. 4, No. 4, 1979, pp 397-434.

[18] S. Abiteboul and R. Hull. IFO: A Formal Semantic Database Model. *ACM Transactions on Database Systems*, Vol. 12, No. 4, 1987, pp 525-565.

[19] R. Hull. Four Views of Complex Objects: A Sophisticate's Introduction. *Nested Relations and Complex Objects in Databases*, LNCS #361, eds. A. Abiteboul, P. C. Fischer, H.-J. Schek, Springer-Verlag, Heidelberg, 1989, pp. 87-116.

[20] M. H. Scholl, H.-J. Schek. A Relational Object Model. *Proceedings of the Third International Conference on Database Theory*, Paris, December 1990, pp.89-105

[21] M. Stonebraker. The Object-Relational DBMSs: The Next Great Wave. Morgan Kaufmann publishers, San Francisco, California, 1996.

[22] M. Tresch and M. H. Scholl. Implementing an Object Model on Top of Commercial Database Systems (Extended Abstract). *Proceedings of the 3rd GI Workshop on Foundation of Database Systems*, Volkse, May 1991.

[23] B. Mitschang, H. Pirahesh, P. Pistor, B. Lindsay, and N. Südkamp. SQL/XNF : Processing Composite Objects as Abstractions over Relational Data. *Proceedings of the 9th Data Engineering Conference*, pp 272--282, Vienna, April 1993. IEEE, IEEE.

[24] J. E. Rumbaugh, M. R. Blaha, W. J. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modelling and Design*. Prentice-Hall International, Inc., 1991.

[25] T. Learmont and R.G.G. Cattell. An Object-Oriented Interface to a Relational Database. In K.R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database Systems*. Springer, 1991.

[26] P. Lyngbaek and W. Kent. A Data Modelling Methodology for the Design and Implementation of Information Systems. In K.R. Dittrich, U. Dayal, and A. P. Buchmann, editors, *On Object-Oriented Database Systems*. Springer, 1991.

[27] I.A. Chen and V.M. Markowitz. OPM Schema Translator 3.1. Technical report LBL-35582, Lawrence Berkeley Laboratory, March 1995.

[28] A. Keller, R. Jensen, S. Agrawal. Persistence Software: Bridging Object Oriented Programming and Relational Databases. *ACM SIGMOD Conference,* 1993, pp 523-528.

[29] M. Rys, M.C. Norrie and H.J. Schek. Intra-Transaction Parallelism in the Mapping of an Object Model to a Relational Multi-Processor System

[30] S. Zdonick and D. Maier. Fundamentals of Object-Oriented Databases. S. Zdonick and D. Maier, editors, Readings in Object-Oriented Database Systems. Morgan-Kaufmann Publishers, Inc., 1990.

[31] R. Ananthanarayanan, V. Gottemkkala, W. Kaefer, T. J. Lehman and H. Pirahesh. Using the Coexistence Approach to Achieve Combined Functionality of Object-Oriented and Relational Systems. *Proceedings ACM SIGMOD International Conference on Management of Data,* Washington, DC, 1993, pp 109-118.

[32] W. Kim, E. Bertino and J.F.Garza. Composite Objects Revisited. *Proceedings of ACM SIGMOD International Conference on the Management of Data*, Portland, Oregon, 1989, pp 337-347.

[33] H.-J. Schek and M. H. Scholl. The Two Roles of Nested Relations in the DASDBS Project. *Nested Relations and Complex Objects in Databases*, LNCS #361, eds. A. Abiteboul, P. C. Fischer, H.-J. Schek, Springer-Verlag, Heidelberg, 1989, pp. 50-68.

[34] R. Lorie and H.-J. Schek. On Dynamically Defined Complex Objects and SQL. *Proceedings of the Second Workshop on Object-Oriented Database Systems*, Bad Münster, September 1988.

[35] L. Kerschberg and J.E.S Pacherco. A Functional Data Base Model. *Technical Report, Pontificia Universidade Catolica do Rio de Janeiro*, Rio de Janeiro, February 1976.

[36] A. Ohori. Representing Object Identity in a Pure Functional Language. *Proceedings of the Third International Conference on Database Theory*, Paris, December 1990, pp.41-55

[37] P. Valduriez. Join Indices. *MCC Technical Report Number DB-052-85*, submitted for publication July 1985.

[38] P. Valduriez, S. Khoshafian and G. Copeland. Implementation Techniques of Complex Objects. *Proceedings of the Twelfth Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, pp.101- 110.

[39] D.S. Batory and A. P. Buchmann. Molecular Objects, Abstract Data Types and Data Models: A Framework. *Proceedings of the Tenth Conference on Very Large Data Bases,* Singapore, 1984, pp 172-184.

[40] B. Mitschang. Extending the Relational Algebra to Capture Complex Objects. *Proceedings of the Fifteenth International Conference on Very Large Databases,* Amsterdam, August 1989, pp 297-305.

[41] W. Kim. A Model of Queries for Object-Oriented Databases. *Proceedings of the Fifteenth International Conference on Very Large Databases,* Amsterdam, August 1989, pp 423-432.

[42] G. M. Shaw and S.B. Zdonik. An Object-Oriented Query Algebra. *IEEE Data engineering*, Vol. 12, No. 3, September 1989, pp 29-36. Special Issue on Database Programming Languages.

[43] S. Abiteboul and N. Bidoit. Non First Normal Form Relations: An Algebra Allowing Data Restructuring. *Journal of Computer Systems Science*, Vol. 33, 1986, pp 361-393.

[44] R. Hull and C. K. Yap. The Format Model: A Theory of Database Organisation. *Journal of the ACM*, Vol. 31, No. 3, 1984, pp 518-537.

[45] R. J. Brachman and J.G. Schmolze. An Overview of the KL-ONE Knowledge Representation System. *Cognitive Science*, Vol. 9, 1985, pp 171-216.

[46] S. Abiteboul and N. Bidoit. Non-First-Normal-Form Relations to Represent Hierarchically Organised Data. *Proceedings of the Third ACM SIGACT/SIGMOD Symposium on the Principles of Database Systems*, 1984, pp 191-200.

[47] P. C. Fisher and S. J. Thomas. Operators for Non-First-Normal-Form Relations. *Proceedings IEEE Computer Software and Applications Conference,* pp 464-475, 1983.

[48] N. LaLiberté. Design and Implementation of a Primary Memory Version of ALDAT, Including Recursive Relations. *McGill University, School of Computer Science, Thesis*. 1986

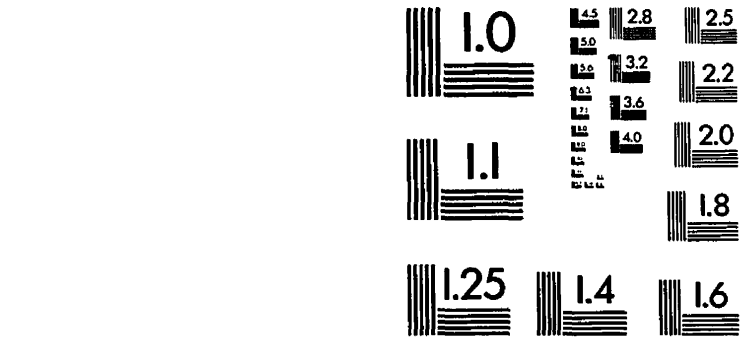
[49] T.H. Merrett. *Relational Information Systems*, Reston Publishing Co., Virginia, 1984.

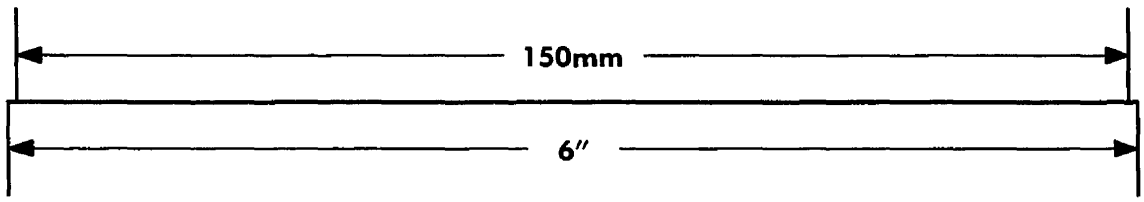
[50] J. Bradley. Application of SQL/N to the Attribute-Relation Associations Implicit in Functional Dependencies. *International Journal of Computer Information Science.* Vol. 12, No. 2, 1983, pp 65-86

[51] D. Chamberlin et al. SEQUEL 2: A Unified Approach to Data Definition, Manipulation and Control. *IBM Journal of Research and Development* Vol. 20, No. 6, pp 560-575 (1976)

[52] Document number ANSI X3.135-1989 entitled "Database Language SQL with Integrity Enhancement"

[53] Document number ISO 9075-1989 entitled "Database Language SQL with Integrity Enhancement"





APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989