Implementation of Nested Relations in a Database Programming Language

Hongbo HE

School of Computer Science
McGill University, Montreal

September 1997

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

Abstract

This thesis discusses the design and implementation of nested relations in Relix, a relational database programming language. The purpose of this thesis is to integrate nested relations into Relix.

While a flat relation is defined over a set of atomic attributes, a nested relation is defined over attributes which can include non-atomic ones, i.e. a data item itself can be a relation. To show the power of relational database systems, it is desirable to have nested relations in Relix. Our implementation was done using existing relational functionalities of Relix, without any modification of the physical data representation. Instead of focusing on nesting and unnesting as the major research direction of nested relations, we built nested relations on top of flat relations and we built nested queries by allowing the domain algebra to subsume the relational algebra.

Users are able to take advantage of nested relations in Relix with only minimal new syntax being added to the system.

Résumé

Cette thèse a pour objectif la spécification et l'implémentation des relations imbriquées dans Relix, un langage de programmation de base de données relationnelles. Le but de cette thèse est d'intégrer les relations imbriquées dans Relix.

Une relation plate est definie sur un ensemble d'attributs atomiques, alors qu'une relation imbriquée est définie sur des attributs qui sont non atomiques,i.e., une donnée pourrait être une relation. Pour montrer la puissance des systems de base de données relationnelles, il est desirable d'avoir des relations imbriquées dans Relix. Notre implémentation est basée sur les fonctionalités relationnelles déjà existantes dans Relix, aucune modification au niveau de la représentation physique des données n'a été apportée. Au lieu de focaliser notre axe de recherche sur les propriétés d'imbrication et de non-imbrication des relations imbriquées, nous avons construit des requêtes imbriquées permettant à l'algèbre relationnelle d'être une composante du domaine algébrique.

Les utilisateurs peuvent tirer profit des relations imbriquées dans Relix à l'aide d'une nouvelle syntaxe minimale qui a été ajoutée au système.

Acknowledgements

I would like to express my gratitude to my thesis supervisor, Professor T. H. Merrett, for his attentive guidance, invaluable advice, and endless patience throughout the research and preparation of this thesis. I would also like to thank him for his financial support.

I would like to thank my colleagues in the ALDAT lab, especially Xiaoyan Zhao and Rebecca Lui for their assistance on the usage of facilities in the lab and their consultation on the existing Relix system. Special thanks goes to Abdelkrim Hebbar who translated the abstract of this thesis to French and Anne Vogt who proofread this thesis.

I would also like to thank all the secretaries of the School of Computer Science for their kind help, especially Ms. Josie Vallelonga and Ms. Franca Cianci.

I wish to thank all my friends during my years at McGill, Pung Hay, Xinan Tang, Shaohua Han and Marcia Cavalcante for their endless encouragement.

Thanks must also go to my father, my brothers for their love and constant support.

Finally, I would like to dedicate this thesis to my mother, for her bless in my life to date and forever.

Contents

Chapter 1

Introduction

The topic of this thesis focuses on the implementation of nested relations in Relix, a relational database system developed at McGill.

The relational model for representing data was proposed by Codd [Cod70] in the early seventies. Since then, it has gained an undisputable key position in the commercial database industry. The nested relational model [Mak77] was developed as an extension of the relational model and has gained significant importance in non-traditional database applications (such as CAD/CAM databases, text and pictorial databases).

1.1 Relational Model

In the relational model, information is represented in a table format with the following properties:

- All rows are distinct from each other.
- The ordering of the rows is unimportant.
- Each column is unique and the ordering of the columns is immaterial.

- The value in each row under a given column is atomic, i.e., it is nondecomposable.

Each row is called a *tuple* and a column is referred to as a *domain*, A name is given to the domain of a relation to release the users from remembering the domain ordering of the relation. They are called *attributes*. From a mathematical perspective, a relation is a subset of the Cartesian product of its *domains*.

1.1.1 Operations on Relations

Operations on relations form the *relational algebra*, and can be thought of as a collection of methods for building new tables that constitute answers to queries. Codd defined a set of relational operations and proved that they are "relationally complete"[1] [Cod72].

Relations are considered atomic objects in the relational algebra, and access to tuples within a relation is precluded. Thus the notation and manipulations that must be done are greatly simplified [Mer84]. The operations are defined as following:

- unary operations
  - projection
  - selection
- binary operations
  - μ-joins: applied to relations that are union compatible
  - σ-joins: support set operations on relations

[1]An algebra or calculus is *relationally complete* if, given any finite collection of relations $R_1, R_2, \ldots, R_n$ in simple normal form, the expressions of the algebra or calculus permit definition of any relation from $R_1, R_2, \ldots, R_n$ by using a set of N range predicates in one-to-one correspondence with $R_1, R_2, \ldots, R_n$.

1.1.2 Operations on Domains

The need for arithmetic and similar processing of the values of attributes in individual tuples is apparent. The *domain algebra* was proposed [Mer76] entirely to avoid tuple-at-a-time operations for processing attributes in individual tuples. It allows the user to create new domains from existing ones, and allows the generation of new values from many values within a tuple or from values along an attribute. The domain algebra operations are defined as:

- horizontal operations
  - Constant
  - Rename
  - Function
  - If-then-else
- vertical operations
  - Reduction
  - Equivalence Reduction
  - Functional Mapping
  - Partial Functional Mapping

1.2 Object Oriented Model

Object-oriented techniques are becoming popular for designing and implementing user interfaces, applications and systems. ODBMS (Object-oriented Database Management System) is the result of objected-oriented techniques implemented in database management systems.

Object-oriented techniques include the following key points:

- Encapsulation: combining data and functions in a single unit, the object.
- Polymorphism: the ability to treat different objects the same way by sending them the same message, which elicits a semantically similar function in each object.
- Class instantiation: creating different objects of the same general description from the same class.
- Inheritance: extending one or more existing objects to create new objects that share data, behavior, and methods in terms of OO terminology.

Generally, ODBMSs are the database systems that allow data to be stored beyond the tabular format of the relational model. They can deal with complex data structures as in programming languages. Another possible way of thinking of ODBMSs is as an object-oriented programming language with persistent data, in the sense that data in the programs lives beyond the life of the programs. The ability to manipulate data and perform computations within one single system is the strong point that has been claimed to solve the problem of the mismatch between data manipulation languages (e.g. SQL) in the relational model and ordinary programming languages.

1.3 Object Relational Model

Another database model is the object-relational database management system, which was proposed by Stonebraker et. al. [Stone96].

It has four major features:

- Support for base data type extension. These include dynamic linking of user-defined functions, client/server activation of user-defined functions, secure user-defined functions, callback in user-defined functions, user-defined access methods, and arbitrary-length data types.

- Support for complex objects. Three basic type constructors are available: composites, sets and references. Full featured user-defined functions can be imposed on complex objects. Complex data types can be of arbitrary-length and have SQL support.
- Support for inheritance. Both data and function inheritance are supported. Overloading is also available, as well as multiple inheritance.
- Support for a production rule system. Events and actions are retrieved as well as updates. Rules are integrated with inheritance and type extension. There are rich execution semantics for rules and no infinite loops.

Stonebraker predicted "object-relational DBMS to be the next great wave in database technology" [Stone96].

1.4 Nested Relation Model

Most work on the relational model of Codd [Cod70] involved the first normal form (1NF) assumption, i.e., that all elements of a tuple of a relation are atomic values (undecomposable). This has the advantage of simplifying the data model. However, from the programming language point of view, this is an arbitrary restriction. Ways of relaxing 1NF have been investigated which retain much of the advantages of the relational model. The need to introduce complex objects into relations to make them more qualified to handle non-business data processing applications such as picture and map processing, computer aided design and scientific applications was realized in the late 1970's, thus leading to the introduction of nested relations [Mak77] and the non-first-normal-form (NF^2) [Jae82].

Project		
Manager	Detail	
	P_Name	Budget(K)
Joe	P1	40
	P2	30
Sue	P2	30
	P3	20
	P4	30

Figure 1.1: Nesting

1.4.1 Nested Relations

The relation *Project* in Figure 1.1 gives an example of nesting. Relation *Project* consists of 2 tuples each having two attributes:

- Manager: The name of the manager who is in charge. The data is of type *string* (atomic).
- Detail: A nested relation containing the projects of which the manager is in charge. Each tuple in relation Detail contains a whole relation as an attribute value. The first tuple contains a relation with 2 tuples. The second tuple contains a relation with 3 tuples.

In [Sch82][Pis86][Lev92], the authors claim that NF^2 relations have some advantages over 1NF relations, such as:

- Nested relations minimize redundancy of data. Related information can be stored in one relation only without redundancy. For example, if relation Project in Figure 1.1 were to be represented by 1NF, it would be either have had to have redundant values for attribute Manager, or it would have had to be split into two different relations (Project and Detail), with a foreign key, P_Name.

- Nested relations allow efficient query processing since some of the joins are realized within the nested relations themselves. In our example in figure 1.1, if information about the manager's budget needs to be retrieved in the 1NF representation a join must be performed between Manager and Detail, while no join is needed in the NF^2 representation.

- Low level implementation techniques such as clustering and repeating fields can be represented using the formalism defined by the nested relation model [Kor89].

1.4.2 Nesting and Unnesting

In the literature, defining a nested relational model was done by extending relational operators to nested relations, and adding two restructuring operators, NEST and UNNEST [Jae82][Fis85]. The NEST operator creates partitions which are based on the formation of equivalence classes [Kor89]. Tuples are equivalent if the values of the same attributes which are not nested are the same in the different tuples. All equivalent tuples are replaced with a single tuple in the resulting relation; the attributes of this tuple consists of all the attributes that are not nested, having the common value in the original tuples, as well as a nested relation whose tuples are the values of the attribute to be nested. Figure 1.2 shows an example of the use of the NEST operator. Relation *Project* is nested on attribute *Member*.

The UNNEST operator undoes the result of the NEST operator. It creates a new relation whose tuples are the concatenation of all the tuples in the relation being unnested to the other attributes in the relation [Kor89]. Thus:

$UNNEST_{Member}(NEST_{Member}(Project))$ = Project [Jae82]

But, the reverse does not hold, i.e.:

Project	
Proj_Name	Member
P1	Joe
P1	Sue
P1	Sam
P2	Joe
P2	Mary
P3	Sue
P3	Mary

$NEST_{Member}$ (Project)	
Proj_Name	Member
P1	Joe
	Sue
	Sam
P2	Joe
	Mary
P3	Sue
	Mary

Figure 1.2: Nesting on Member

R	
A	B
x	a
	b
x	a
	c

$R' = UNNEST_B R$	
A	B
x	a
x	b
x	c

$R'' = NEST_B R'$	
A	B
x	a
	b
	c

Figure 1.3: $NEST_B(UNNEST_B(R)) <> R$

"$NEST_{Attribute}(UNNEST_{Attribute}(Relation))$ = Relation" is not always true. The case in Figure 1.3 gives an example.

As the price of the advantages over 1NF relations, nested relations pose a non-trivial problem of data representation [Tak89]. There are generally alternative representations of data in a nested relation, while the data is uniquely represented by a 1NF relation. This is illustrated by the following example:

In left side of Figure 1.2, we have a simple 1NF relation *Project* on *Proj_Name* and *Member*. This relation is a unique representation of a set of 7 tuples.

<table>
<tr><th colspan="2">$NEST_{Proj\_Name}(Project)$</th></tr>
<tr><th>Proj_Name</th><th>Member</th></tr>
<tr><td>P1</td><td rowspan="2">Joe</td></tr>
<tr><td>P2</td></tr>
<tr><td>P2</td><td rowspan="2">Mary</td></tr>
<tr><td>P3</td></tr>
<tr><td>P1</td><td rowspan="2">Sue</td></tr>
<tr><td>P3</td></tr>
<tr><td>P1</td><td>Sam</td></tr>
</table>

Figure 1.4: Relation: $NEST_{Proj\_Name}(Project)$

We can nest *Project* on attribute *Member* as shown in the right side of Figure 1.2. We can also nest *Project* on attribute *Proj_Name*, as illustrated in Figure 1.4.

Thus, it might be controversial whether or not these two relations are regarded as the same relation. There are two different assumptions with respect to the interpretation [Tak89]:

1. To consider each tuple in the relation to be meaningful. Hence, the relation in the right side of FigureΦ 1.2 gives a list of projects and their members, while the relation in FigureΦ 1.4 gives the list of members and the projects they participate. They carry different meanings, therefore, each nested relation should be recognized as distinct. Thus, it would be difficult to identify a nested relation with a 1NF relation. It "poses a semantic gap between 1NF and nested form relations although it enables us to represent complex objects in a natural way by using nested relations" [Tak89].

2. Conversely, to assume that each tuple is just a union of single values rather than a specific object, which allows the identification of the two nested relations

in the right side of Figures 1.2 and 1.4 and the identification of them with the original 1NF relation. Many research papers implicitly use this assumption such as those proposing transformation operators [Jae82][Fis85], and those designing nested relations [Ozy87][Ozy89].

Significant progress has been made in the field of nested relations during the past decade. A generalization of the ordinary relational model, allowing relations with set-valued attributes and adding two restructuring operators, nest and unnest, was introduced [Jae82][OOM87]. Fisher and Van Gucht [Fis85] discussed one-level nested relations and their characterization by a new family of dependencies, and furthermore, they developed a polynomial-time algorithm to test if a structure is a one-level nested relation. Thomas and Fischer generalized their work on the one-level model and allowed nested relations of arbitrary, but fixed depth [Tho86]. In [RKS86], Roth, Korth and Silberschatz defined a normal form called "Partitioned Normal Form(PNF)" for nested relation, and also defined algebra and calculus query languages for them; however, their proofs and method were later questioned by Tansel and Garnett [Tag92]. Numerous query languages have been introduced for the nested model [RKS86], and extensions have been proposed to practical query languages such as SQL to accommodate nesting [Pis86][Kor89]. Implementation of databases based on the nested relation model are also available such as of in [Sps87][Des88][Sab89]. These are either built on top of existing relational databases, or from scratch.

1.4.3 Our Approach

We view nested relations in a different light. We do not restrict our approach to nesting and unnesting. We build nested relations to facilitate nested queries. We do this by extending domain operations to include relational operations.

In our approach, we observe that:

- Using flat relations, we can model nested relations. We can use a set of surrogates to keep links between parent relations and their nested child relations.
- We can build a nested relation query facility in the context of flat relations. Since an attribute itself can be a relation, relational operations can be included in domain operations.

1.5 Thesis Aim and Outline

The purpose of this thesis is to extend Relix with nested relations and to integrate the relational algebra into the domain algebra.

- Chapter 1 contains a literature review of the relational model, the object oriented model, object-relational model and nested relations.
- Chapter 2 provides a general overview of the Relix database programming language—the relational database programming language developed at McGill University. The syntax and internal operation of Relix that are relevant to the work done in this thesis are discussed in this chapter.
- Chapter 3 is the user's manual on nested relations. It shows the semantics and syntax for nested relation definitions and operations.
- Chapter 4 gives a detailed description of the implementation of nested relations in Relix.
- Chapter 5 concludes the thesis with a summary and proposals for future work.

Chapter 2

Relix

Relix is briefly described in this Chapter. The purpose of this Chapter is to provide readers with enough background to understand the rest of the thesis. Since all the design and implementation work in this thesis follows the conceptual framework of the existing Relix system, we will present only the subset of Relix related to this thesis. The theoretical foundation on which the development of Relix is based can be found in [Mer84], while the basic reference of Relix can be found in [Lal86].

2.1 Overview

Relix is a **RE**lational database programming **L**anguage in UN**IX**. It is an interpreted language written in C. It can accept and execute commands or statements from the command line. It can also accept Relix commands and statements batch files.

Relix deals primarily with two kinds of data models: domains and relations. There are two categories of operations: domain algebra and relational algebra.

2.1.1 Domains and Relations

A relation is defined on one or more attributes, and the data for a given attribute is from a particular domain of values. The domain of a given attribute determines its data type.

For example the *Student* relation in Figure 2.1 is defined on four attributes: *Stu_id*, *Enter_year*, *Name*, *Canadian*. The domains of *Stu_id* and *Enter_year* attributes are integer. The domain of *Name* attribute is string. And the domain of *Canadian* attribute is boolean.

```
Student
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Stu_id          Enter_year      Name    Canadian
- - - - - - - - - - - - - - - - - - - - - - - - - - -

9546900         1995            Joe     true
9602324         1996            Sue     true
9701087         1997            Jin     false
9702340         1997            Jin     false
- - - - - - - - - - - - - - - - - - - - - - - - - - -
```

Figure 2.1: Student relation

There are six atomic data types in Relix as shown in Figure 2.2. Note that we also have a special data type, *relation*, which will be introduced in Chapter 3.

In Relix, we can declare the domains of relation *Student* as follows:

> **domain** *Stu_id* **integer ;**

> **domain** *Enter_year* **integer ;**

> **domain** *Name* **string ;**

> **domain** *Canadian* **boolean ;**

The relation *Student* can then be declared and initialized:

Data Type	Short Form	Domain
integer	int	singed integer
long	long	signed long integer
short	short	sighed short integer
real	real	sighed floating point
string	strg	sequence of characters (with limitations)
boolean	bool	true or false

Figure 2.2: Atomic Data Type in Relix

> **relation** *Student(Stu_id, Enter_year, Name, Canadian)* < −
{(9546900, 1995, "Joe ", true),
(9602324, 1996, "Sue ", true),
(9701087, 1997, "Jin ", false),
(9702340, 1997, "Jin ", false)} ;

We can also declare a relation without initialization, i.e., a relation without any data :

> **relation** *Student(Stu_id, Enter_year, Name, Canadian)*

2.1.2 Basic Commands in Relix

In Relix, there are basic commands to show, print and delete domains and relations declared in the database.

The grammar for the commands is:

<command_name> (**!** or **!!**<parameters>).

Where <command_name> includes reserved words which will be introduced in the following paragraphs and **!** means that the programmer is prompted for the parameters, while **!!** requires command line parameters.

Show Commands

- **sd!** or **sd!!**<domain_name>

  Relix will show the name, type and other information associated with all domains in the database or the specified domain. For example:

  > **sd!!** *Stu_id*

  will show the information of domain *Stu_id*.

- **sr!** or **sr!!**<relation_name>

  Relix will show the name, degree and other information of all relations in the database or the specified relation. For instance:

  > **sr!!** *Student*

  will show the information of relation *Student*.

- **srd!** or **srd!!**<relation_name>

  Relix will show all relations and their domains in the database or the specified relation and its domains. For example:

  > **srd!!** *Student*

  will show relation *Student* and its domains.

- **pr!!**<relation_name>

  Relix will print all data in the specified relation. For instance:

  > **pr!!** *Student*

  will print all data in relation *Student*.

- **dd!!**<domain_name>

  Relix will delete the specified domain. If it is still in use, Relix will give an error message and the domain will not be deleted.

  > **dd!!** *Year*

  will delete domain *Year*, if it is not in use.

- **dr!!**<domain_name>

  Relix will delete the specified relation.

  > **dd!!** *Student*

  will delete relation *Student*.

- **q!**

  This command can be used to quit the Relix system.

2.2 Relational Algebra

The relational algebra consists of a set of operations on relations. Both operands and results are relations.

In Relational Algebra operations, we have unary operations and binary operations. As the names indicate, unary operators take one relation as an operand, and binary operators take two relations as operands. In unary operations, there are projection and selection; in binary operations, there are joins.

2.2.1 Projection

Projection is an operation on the attributes of a given relation. The results of a projection is a relation whose attributes are the specified attributes in the projection list. Duplicate tuples in the resulting relation are removed. For example, we can project the *Name* of *Student* relation as follows:

$> \quad Stu\_name \quad <- [\ Name\ ]\ \textbf{in}\ Student\,;$

```
Stu_name
- - - - -
Name
- - - - -
Jin

Joe

Sue
```

2.2.2 Selection

Selection is an operation on a relation to select tuples that meet the condition specified in the selection clause, which is called *T-selector*(tuple selector). We can do the following selection to extract the student information about who is a Canadian.

$> \quad Ca\_stu \quad <- \textbf{where}\ Canadian = true\ \textbf{in}\ Student\,;$

or

$> \quad Ca\_stu \quad <- \textbf{where}\ Canadian\ \textbf{in}\ Student\,;$

```
Ca_Stu
- - - - - - - - - - - - - - - - - - - - - - - - - - -
Stu_id        Enter_year     Name    Canadian
- - - - - - - - - - - - - - - - - - - - - - - - - - -

9546900       1995           Joe     true
9602324       1996           Sue     true
- - - - - - - - - - - - - - - - - - - - - - - - - - -
```

We can combine projection and selection in a single statement. First Relix will do selection on the input relation based on the selection clause, then do projection on the output of the selection. We can extract the Stu_id numbers of students who are Canadian using the following statement:

$> \quad Ca\_stu\_id \quad <- [\ Stu\_id\ ]$ **where** $Canadian$ **in** $Student$;

```
Ca_stu_id
------
Stu_id
------
9546900
9602324
```

2.2.3 Joins

There are two classes of join operations in Relix: μ-joins, the family of set-valued set operations; and σ-joins, the family of logical-valued set operations [Mer84].

μ-joins

μ-joins are derived from the set operators such as intersection, union, difference, etc. The μ-joins on two relations, R(X,Y) and S(Y,Z), are based on three parts:

- *center* $\triangleq \{(x, y, z) \mid (x, y) \in R$ **and** $(y, z) \in S\}$
- *left wing* $\triangleq \{(x, y, DC) \mid (x, y) \in R$ **and** $\forall\, z(y, z) \notin S\}$
- *right wing* $\triangleq \{(DC, y, z) \mid (y, z) \in S$ **and** $\forall\, x(x, y) \notin R\}$

We will explain these three basic μ-joins in detail in this section. The two relations in Figure 2.3 are used to illustrate the operations:

- The most used μ-join is the natural join (*ijoin* or *natjoin*), which gives us the *center* part of the operand relations. It combines tuples of the two relations that have equal values on the join attributes. Thus, it is the intersection of the two relations on the join attributes, which gives us *ijoin*.

Student

Stu_id	Name
9546900	Joe
9602324	Sue
9701087	Jin
9702340	Jin

Courses

Stu_id	c_name
9576701	Math
9546900	Physics
9602324	History
9602324	Math

Figure 2.3: Student and Courses relations

The natural join of relations R and S is defined as [Cod70]:

R **natjoin** $S \triangleq \{(a, b, c) \mid R(a, b)$ **and** $S(b, c)\}$

where (a,b,c) is a tuple in the new relation, of which (a,b) is a tuple of R and (b,c) is a tuple of S.

The following Relix statement performs a *natjoin* between relation Student and relation Courses.

$>$ *SijoinC* $<-$ *Student* **ijoin** *Courses* ;

SijonC

Stu_id	Name	C_Name
9546900	Joe	Physics
9602324	Sue	History
9602324	Sue	Math

- The union join (**ujoin**) is an operation that is a union of the set of tuples from the natural join, together with the tuples from the relations of both sides that are not equal to each other in the join attributes, and the missing attributes are filled up with DC[1] null value. It gives us the union of the *left, center, right* parts of the operand relations.

$> \quad SujoinC \quad < - \ Student$ **ujoin** $Courses$;

```
SujoinC
- - - - - - - - - - - - - - - - - -
Stu_id         Name    C_Name
- - - - - - - - - - - - - - - - - -
9546900        Joe     Physics
9576701        DC      Math
9602324        Sue     History
9602324        Sue     Math
9701087        Jin     DC
9702340        Jin     DC
- - - - - - - - - - - - - - - - - -
```

- The symmetric difference join (**sjoin**) is the set of tuples from the relations of both sides that are not equal to each other in the join attributes, the missing attributes are filled up with DC null value. It gives us the union of the *left, right* parts of the operand relations.

$> \quad SsjoinC \quad < - \ Student$ **sjoin** $Courses$;

```
SsjoinC
- - - - - - - - - - - - - - - - -
Stu_id         Name    C_Name
- - - - - - - - - - - - - - - - -
9576701        DC      Math
9701087        Jin     DC
9702340        Jin     DC
- - - - - - - - - - - - - - - - - -
```

The overall μ-join operations are shown in Figure 2.4.

[1]DC, Don't Care, describes irrelevant values.

μ–joins	μ–join-operator	Resulting Relation
Natural Join	‘natjoin’ or ‘ijoin’	centre
Union Join	‘ujoin’	left U centre U right
Left Join	‘ljoin’	left U centre
Right Join	‘rjoin’	right U centre
Left Difference Join	‘djoin’ or ‘dljoin’	left
Right Difference Join	‘drjoin’	right
Symmetric Difference Join	‘sjoin’	left U right

Figure 2.4: μ-join operations

σ-joins

The family of σ-joins are based on set comparison operators. In operations, the tuples in each of the operand relations are grouped such that for each group, all the non-join attributes on both sides are identical. The set comparison operator is then applied to the Cartesian product of the groups. The values of the non-join attributes of the comparing groups are accepted if the specified set comparison on the join attributes is satisfied.

There are five σ-joins:

- **sup** or **div** or **gejoin**, the superset operator, a generalization of $\supseteq$. 'div' stands for 'division', which extends Codd's definition of relational *division* [Cod72].
- **sub** or **lejoin**, subset, a generalization of $\subseteq$.
- **eqjoin**, equal set, a generalization of $=$.
- **sep**, intersection empty, a generalization of $\Cap$.
- **icomp**, intersection not empty, a generalization of $\not\Cap$.

Considering the two relations *Student* and *Class* in Figure 2.5.

Student

Name	Course
Joe	Math
Joe	Physics
Sue	Physics
Jin	Math

Class

Course	Room
Math	286
Physics	286
Chemistry	302
Physics	312

Figure 2.5: Student and Class relations

To answer following query: Find students and the classrooms such that the courses the student has taken is a subset of the courses which are given in this classroom.

$> \quad StuRoom \quad <- Student \; \mathbf{sub} \; Class;$

```
StuRoom
- - - - - - - - -
Name    Room
- - - - - - - - -
Joe     286
Jin     286
Sue     286
- - - - - - - - -
```

The overall σ-join operations are shown in Figure 2.6.

2.3 Domain Algebra

Relational algebra considers relations to be data primitives [Mer84] and therefore does not give the user the power to manipulate attributes. To overcome this problem, Merrett proposed domain algebra [Mer77].

Besides creating a domain by declaring its type as in section 2.1.1, one can build a new domain by expressing the domain as operation on existing domains. It allows operations over a single tuple (horizontal operations) and operations over sets of tuples (vertical operations). Domains defined in this way are 'virtual' in the sense that they are expressions and no actual values are associated with them. The values of the virtual domains are actualized in a Relix statement, notably, projection or selection.

2.3.1 Horizontal Operations

Horizontal operations work on a single tuple of relation. We can define constants, perform renaming and arithmetic functions, as well as if-then-else expressions.

σ-joins	Set Comparison	σ-join Operator
$\supseteq$	Superset	‘div’ or ‘sup’ or ‘gejoin’
$=$	Equal Set	‘eqjoin’
$\subseteq$	Subset	‘sub’ or ‘lejoin’
$\cap\!\!\!\!{\scriptstyle 0}$	Intersection Empty	‘sep’
$\supset$	Proper Superset	‘gtjoin’
$\subset$	Proper Subset	‘ltjoin’
$\not\supseteq$	Not Superset	‘~sup’
$\neq$	Not Equal Set	‘~eqjoin’
$\not\subseteq$	Not Subset	‘~sub’
$\not\cap\!\!\!\!{\scriptstyle 0}$	Intersection Not Empty	‘icomp’
$\not\supset$	Not Proper Superset	‘~gtjoin’
$\not\subset$	Not Proper Subset	‘~ltjoin’

Figure 2.6: σ-join operations

- constants

  let two **be** 2;

  let myname **be** "marc";

- renaming

  let stu_name **be** name;

- arithmetic functions

  let Sin **be** sin(angle);

  let area **be** sqrt(a**2 + b**2 + c**2) / 2;

- if-then-else

  let Grade **be if** Mark > 60 **then** "Pass" else "Fail";

All above domains defined are virtual domains. For example, we can actualize *Grade* as following:

> *GRADES* < − [*Student, Grade*] **in** *MARKS*

```
MARKS                  GRADES
- - - - - - - -        - - - - - - - - -
Name   Mark            Name   Grade
- - - - - - - -        - - - - - - - - -
Joe    50              Joe    Fail
Jin    80              Jin    Pass
Sue    90              Sue    Pass
- - - - - - - -        - - - - - - - - -
```

2.3.2 Reduction (Vertical Operations)

Reduction consists of domain algebra operations which combine values from more than one tuple – the 'vertical' operation [Mer84].

- Simple Reduction

Simple reduction produces a single result from the values from all tuples of a single attribute in the relation [Mer84]. The operator in simple reduction must be both *commutative* and *associative*, such as plus (+), multiplication (*). For example:

```
let Total be red + of Grade;
```

```
Transcript
------------
Name   Dept   Grade        (Total)
------------
Joe    CS     85             330
Jin    CS     90             330
Sue    EE     80             330
Weny   ME     75             330
------------
```

- Equivalence Reduction

Equivalence reduction is like simple reduction but produces a different result from different sets of tuples in the relation. Each set is characterized by all tuples having the same value for some specified attributes – an "equivalence class" in mathematical terminology [Mer84]:

```
let Subtotal be equiv + of Grade by Dept;
```

```
Transcript
------------
Name   Dept   Grade        (Subtotal)
------------
Joe    CS     85             175
Jin    CS     90             175
Sue    EE     80             80
Weny   ME     75             75
------------
```

2.3.3 Nested Relations

In this thesis, we extend Relix to support nested relations. In chapter 3 and chapter 4, we will discuss nested relations in detail, including a user manual and implementation techniques.

2.4 *ijoin, ujoin, sjoin* are Associative and Commutative

From Section 2.3.2, we know that in simple and equivalence reduction, the operator needs to satisfy the commutative and associative criteria. In the following sections, we prove that *ujoin*, *ijoin*, and *sjoin* all have these two characteristics .

2.4.1 Definition

For relations, R(X, Y) and S(Y,Z), these three sets of tuples are each defined on the attributes(or attribute groups) X, Y, Z.

We first define three disjoint sets of tuples which are set operations between R and S [Mer84]:

1. $center \triangleq \{(x, y, z) \mid (x, y) \in R \textbf{ and } (y, z) \in S\}$
2. $left\ wing \triangleq \{(x, y, DC) \mid (x, y) \in R \textbf{ and } \forall\, z(y, z) \notin S\}$
3. $right\ wing \triangleq \{(DC, y, z) \mid (y, z) \in S \textbf{ and } \forall\, x(x, y) \notin R\}$

The joins' definitions are based on these 3 sets:

1. R **ijoin** S $\triangleq$ *center*
2. R **ujoin** S $\triangleq$ *left wing* $\cup$ *center* $\cup$ *right wing*
3. R **sjoin** S $\triangleq$ *left wing* $\cup$ *right wing*

2.4.2 Commutative

By definition, an binary operator θ is commutative *iff* $A\ \theta\ B\ =\ B\ \theta\ A$.

<u>*Remark* 1</u>: R **ijoin** S = S **ijoin** R.

Proof:

R **ijoin** S = $\{(x,y,z) \mid (x,y) \in R$ **and** $(y,z) \in S\}$ (from definition)

$\Longrightarrow$

R **ijoin** S = $\{(z,y,x) \mid (z,y) \in S$ **and** $(y,x) \in R\}$ (from the commutativity of **and**)

$\Longrightarrow$

R **ijoin** S = S **ijoin** R

<u>*Remark* 2</u>: R **sjoin** S = S **sjoin** R.

Proof:

R **sjoin** S = $\{(x,y,DC) \mid (x,y) \in R$ **and** $\forall\, z(y,z) \notin S\} \cup \{(DC,y,z) \mid (y,z) \in S$ **and** $\forall\, x(x,y) \notin R\}$ (from definition)

$\Longrightarrow$

R **sjoin** S = $\{(z,y,DC) \mid (z,y) \in S$ **and** $\forall\, x(y,x) \notin S\} \cup \{(DC,y,x) \mid (y,x) \in R$ **and** $\forall\, z(z,y) \notin S\}$ (from symmetry and the commutativity of $\cup$)

$\Longrightarrow$

R **sjoin** S = S **sjoin** R

<u>*Remark* 3</u>: R **ujoin** S = S **ujoin** R.

Since R **ujoin** S = (R **ijoin** S) $\cup$ (R **sjoin** S) (from the definition)

And from Remark 1 and Remark 2, the proof is trivial.

2.4.3 Associative

By definition , an binary operator θ is associative *iff* $(A\ \theta\ B)\ \theta\ C\ =\ A\ \theta\ (B\ \theta\ C)$

Suppose we have 3 relations, R(X,Y), S(Y,Z), T(Z,W)

<u>*Remark* 4</u>: (R **ijoin** S) **ijoin** T = R **ijoin** (S **ijoin** T)

Proof:

(R **ijoin** S) **ijoin** T = $\{(x, y, z) \mid (x, y) \in R$ **and** $(y, z) \in S\}$ **ijoin** T (from the definition)

$\Longrightarrow$

(R **ijoin** S) **ijoin** T = $\{(x, y, z, w) \mid (x, y) \in R$ **and** $(y, z) \in S$ **and** $(z, w) \in T\}$ (from the definition)

$\Longrightarrow$

(R **ijoin** S) **ijoin** T = $\{(x, y, z, w) \mid (x, y) \in R$ **and** $((y, z) \in S$ **and** $(z, w) \in T)\}$ (from the associativity of and)

$\Longrightarrow$

(R **ijoin** S) **ijoin** T = R **ijoin** $\{(y, z, w) \mid (y, z) \in S$ **and** $(z, w) \in T\}$ (from definition)

$\Longrightarrow$

R **ijoin** S) **ijoin** T = R **ijoin** (S **ijoin** T) (from definition)

Remark 5: (R **sjoin** S) **sjoin** T = R **sjoin** (S **sjoin** T)

Proof:

(R **sjoin** S) **sjoin** T = ($leftwing_{(R,S)} \cup rightwing_{(R,S)}$) **sjoin** T (from definition)

$\Longrightarrow$

(R **sjoin** S) **sjoin** T = $(\{(x, y, DC) \mid (x, y) \in R$ **and** $\forall\, z(y, z) \notin S\} \cup \{(DC, y, z) \mid (y, z) \in S$ **and** $\forall\, x(x, y) \notin R\})$ **sjoin** T (from definition)

$\Longrightarrow$

(R **sjoin** S) **sjoin** T = $\{(x, y, DC, DC) \mid (x, y) \in R$ **and** $\forall z(y, z) \notin S$ **and** $\forall w(DC, w) \notin T\} \cup \{(DC, y, z, DC) \mid (y, z) \in S$ **and** $\forall\, x(x, y) \notin R$ **and** $\forall\, w(z, w) \notin T\} \cup \{(DC, DC, z, w) \mid (z, w) \in T$ **and** $\forall\, y(y, z) \notin S$ **and** $\forall\, x(x, DC) \notin R\}$ (from definition)

In the same way, we can get:

R **sjoin** (S **sjoin** T) = $\{(x, y, DC, DC) \mid (x, y) \in R$ **and** $\forall z(y, z) \notin S$ **and** $\forall w(DC, w) \notin T\} \cup \{(DC, y, z, DC) \mid (y, z) \in S$ **and** $\forall\, x(x, y) \notin R$ **and** $\forall\, w(z, w) \notin T\} \cup$

$\{(DC, DC, z, w) \mid (z, w) \in T$ **and** $\forall\, y(y, z) \notin S$ **and** $\forall\, x(x, DC) \notin R\}$

Thus

(R **sjoin** S) **sjoin** T = R **sjoin** (S **sjoin** T)

Remark 6: (R **ujoin** S) **ujoin** T = R **ujoin** (S **ujoin** T)

Proof: From Remark 4 and Remark 5, the proof of Remark 6 is trivial.

2.4.4 Another Approach

Let x be a tuple, and let X be a binary variable such that if x $\in$ some relation R, then X has value 1, otherwise 0.

1. for R = R_1 **ujoin** $R_2 \ldots R_n$ and for some tuple x, if $X_1 + X_2 + \ldots + X_n = 1$, $\Longrightarrow$ x $\in$ R.[2]

2. for R = R_1 **ijoin** $R_2 \ldots R_n$ and for some tuple x, if $X_1 * X_2 * \ldots * X_n = 1$, $\Longrightarrow$ x $\in$ R. [3]

3. for R = R_1 **sjoin** $R_2 \ldots R_n$ and for some tuple x, if $X_1 \oplus X_2 \oplus \ldots \oplus X_n \oplus =$ 0, $\Longrightarrow$ x $\in$ R. [4]

   From characteristics of $\oplus$, we can conclude that if x appears odd times in relations $R_1 \ldots R_n$, then x $\in$ R.

[2] Here + means logical operation OR, which is commutative and associative

[3] Here $*$ means logical operation AND, which is commutative and associative

[4] Here $\oplus$ means logical operation XOR, which is commutative and associative

Chapter 3

User's Manual on Nested Relations

This chapter describes how to define and manipulate nested relations in Relix. Section 3.1 explains the basic concept of nested relations in Relix and presents the initialization of nested relations. Section 3.2 illustrates the operations that can be imposed on nested relations.

3.1 The Nested Relations and Relation Data Type

To introduce nested relations, we add a *relation* data type to Relix. The operations imposed on it are those relational operations on regular relations with some limitations.

We will show an example first, then we will explain how to declare and initialize nested relations, and finally we explain the internal data representations.

> **domain** *A* **intg**;

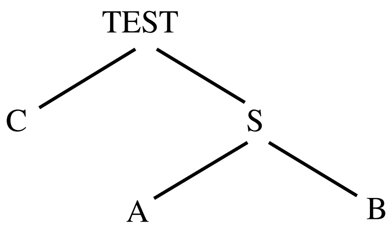
> **domain** *B* **intg**;

> **domain** *C* **intg**;

> **domain** *S* (*A*,*B*);

> **relation** *TEST* (*C*, *S*) <- {(*3*,{(*1*,*2*),(*8*,*7*)}),(*7*,{(*6*,*5*),(*4*,*9*)})};

The above Relix commands are used to initialize the sample nested relation in Figure 3.1.



TEST		
C	S	
	A	B
3	1	2
	8	7
7	6	5
	4	9

Figure 3.1: Sample nested relation: schema tree and value table

We have three regular domains A, B and C, which are defined as integers, and a nested domain S, which is defined upon A and B. When we declare $TEST$, it includes the nested domain S. Relix will consider S as a domain as well as a relation.

The data in S is stored in another relation outside the parent relation $TEST$, which has the same name as S. References to the data (called RELATION .id) are stored in attribute S of relation $TEST$. However, this method of implementation is largely transparent to users, who manipulate the attributes of nested domains as if the data were stored directly in the parent relations.

> **pr!!***TEST*

```
-------------------------------------------------------------------
C        S
-------------------------------------------------------------------
3       0
7       1
-------------------------------------------------------------------
relation: "TEST" has "2" tuple(s)

-------------------------------------------------------------------
.id      A         B
-------------------------------------------------------------------
0       1       2
0       8       7
1       4       9
1       6       5
-------------------------------------------------------------------
relation: "S" has "4" tuple(s)
```

Figure 3.2: What is shown in Relix

Any Relix operation that displays an attribute of type *RELATION* will display the attribute as a number. The actual data of the attribute is printed below it as a separate relation whose *.id* field links it to its parent. In above *print* command, *TEST* and its nested domain *S* are printed out. In child relation *S*, *.id* is mapped to attribute *S* of *TEST*.

The formal syntax of declaration and initialization is as follows:

<declaration>	:=	'domain' <domain_name> '(' <attribute_list> ')'
<initialization>	:=	'relation' <relation_name> '(' <attribute_list> ')'
	:=	< − <tuple_list>

Note in the following sections, we will use the conceptual format as shown in Figure 3.1 to show the example, while in Relix, the actual format will be as in *pr!!*, i.e. as shown in Figure 3.2.

So far, we have only implemented two levels of nesting. Future work is needed to gain multiple level nesting.

3.2 Operations on Nested Relations

In this section, we will show by example how to conduct operations on nested relations. We will show vertical operations, followed by horizontal operations.

The schema of nested relation is represented by the *schema tree* [Ozy87], as shown in Figure 3.3. The nested relation schema of the *Faculty of Engineering* database is: *Dept*, *Building*, *Professor* and *Secretary*, in which *Dept* and *Building* are regular simple domains, and *Professor* and *Secretary* are nested domains, which are further defined by *Name*, *Salary* and *Commit*.

The nested relation, *FactEng*, over the schema tree of Figure 3.3, is shown in Figure 3.4.

3.2.1 Vertical Operations

This section is for the purpose of extending reductions (vertical operations) from scalar attributes to nested relation attributes.

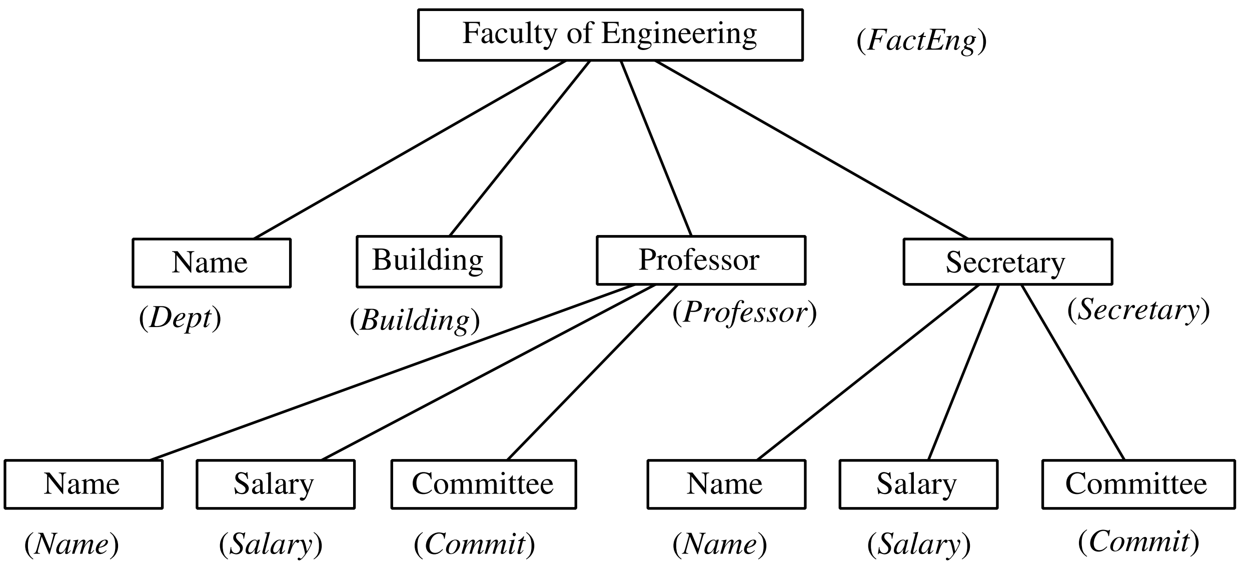


Figure 3.3: The schema tree of the sample

FactEng							
Dept	*Building*	*Professor*			*Secretary*		
		Name	*Salary*	*Commit*	*Name*	*Salary*	*Commit*
CS	MC	Pat	65	PADS	Sal	35	PODS
		Paul	55	PODS	Sue	38	PODS
		Pully	50	SIGM			
EE	MC	Pat	65	PADS	Sandy	36	IEEE
		Paul	55	PODS	Sharon	35	PODS
		Piree	54	IEE	Sam	40	PODS
ME	MD	Pat	65	PADS	Sandra	35	MEE
		Ping	57	MEE	Syl	37	MDS

Figure 3.4: The nested relation, *Engineering Department*, over the schema in Fig.3.3

Simple Reduction

Recall that we already proved that *ijoin*, *ujoin* and *sjoin* are all *commutative* and *associative* (see Section 2.4), we can now extend the reduction operations to *ijoin*, *ujoin*, and *sjoin*.

We start with the following example: Suppose we want to find all the professors in the faculty of engineering, we can do the following query:

> **let** *EngProf* **be red ujoin of** *Professor*

> *AllEngProf* < − [*EngProf*] **in** *FactEng*

> **pr!!** *AllEngProf*

AllEngProf		
EngProf		
Name	*Salary*	*Commit*
Pat	65	PADS
Paul	55	PODS
Piree	54	IEE
Ping	57	MEE
Pully	50	SIGM

Figure 3.5: All Professors of Faculty of Engineering

The formal syntax of simple reduction is as follows:

<simple_reduction_statement>	:=	'let' <new_nested_domain_name> 'be red'
		<binary_operator> 'of'
		<nested_domain_name>
<binary_operator>	:=	'ijoin' | 'ujoin' | 'sjoin'

Now we introduce the *universal professor*, who works in every unit of an education organization.

Query: Find all the universal engineering professors.

\> **let** *UnivEngProf* **be red ijoin of** *Professor*

\> *UEP* $<-$ [*UnivEngProf*] **in** *FactEng*

\> **pr!!** *UEP*

UEP		
UnivEngProf		
Name	*Salary*	*Commit*
Pat	65	PADS

Figure 3.6: All universal engineering professors

If we do *sjoin* on the attribute *Professor*, we obtain professors who are assigned an odd number of positions (see Section 2.4.4 for explanation). Thus we have the following query:

Find all the engineering professors who are assigned an odd number of positions.

\> **let** *OddProf* **be red sjoin of** *Professor*

\> *OProf* $<-$ [*OddProf*] **in** *ED*

\> **pr!!** *OProf*

OProf		
OddProf		
Name	*Salary*	*Commit*
Pat	65	PADS
Ping	57	MEE
Piree	54	IEE
Pully	50	SIGM

Figure 3.7: Professors with an odd number of positions

Equivalence Reduction

Like simple reduction, equivalence reduction is extended to *ujoin*, *ijoin* and *sjoin* as well.

Query: Find the professors by each building.

> $>$ **let** *ProfbyBuild* **be equiv ujoin of** *Professor* **by** *Building*
>
> $>$ *PbB* $<-$ [*Building, ProfbyBuild*] **in** *FactEng*
>
> $>$ **pr!!** *PbB*

PbB			
Building	*ProfbyBuild*		
	Name	*Salary*	*Commit*
MC	Pat	65	PADS
	Paul	55	PODS
	Piree	54	IEE
	Pully	50	SIGM
MD	Pat	65	PADS
	Ping	57	MEE

Figure 3.8: Professors in each building

Query: Find the universal professors by building. (we introduced the idea of a universal professor in the last section. Here a universal professor in each building works in each department of the building)

> $>$ **let** *UnivBuilProf* **be equiv ijoin of** *Professor* **by** *Building*
>
> $>$ *UBP* $<-$ [*Building, UnivBuilProf*] **in** *FactEng*
>
> $>$ **pr!!** *UBP*

UBP			
Building	*UnivBuildProf*		
	Name	*Salary*	*Commit*
MC	Pat	65	PADS
	Paul	55	PODS
MD	Pat	65	PADS
	Ping	57	MEE

Figure 3.9: Universal Professors in each Building

Query: Find the professors in each building who are assigned odd department positions in that building.

> **let** *OddBuilProf* **be equiv sjoin of** *Professor* **by** *Building*

> *OBP* $<-$ [*Building, PureBuilProf*] **in** *FactEng*

> **pr!!** *OBP*

OBP			
Building	*PureBuilProf*		
	Name	*Salary*	*Commit*
MC	Piree	54	IEE
	Pully	50	SIGM
MD	Pat	65	PADS
	Ping	57	MEE

Figure 3.10: Professors who are assigned odd positions in the building

Syntax:

<equiv_reduction_statement> := 'let' <new_nested_domain_name> 'be' 'equiv'
<binary_operator> 'of' <nested_domain_name>
'by' <attribute_list>
<binary_operator> := 'ijoin' | 'ujoin' | 'sjoin'

3.2.2 Horizontal Operations

Horizontal operations consists of binary operations and general operations.

Binary Operations

Binary relational operations take two relations as operands and produce a relation as a result. We extend those operations to nested domains, and take two nested domains as operands and produce a nested domain as a result, which itself is a relation data type.

Query: Find all the staff of the faculty of engineering.

\> **let** *Staff* **be** *Professor* **ujoin** *Secretary*
\> *FactEngStaff* < − [*Dept, Building, Staff*] **in** *FactEng*
\> **pr!!** *FactEngStaff*

The result is in Figure 3.11.

The formal syntax is as follows:

<binary_statement> := 'let' <new_nested_domain_name> 'be'
<nested_domain_name> <binary_operator>
<nested_domain_name>
<binary_operator> := 'ijoin' | 'ujoin' | 'sjoin'

FactEngStaff				
Dept	*Building*	*Staff*		
		Name	*Salary*	*Commit*
CS	MC	Pat	65	PADS
		Paul	55	PODS
		Pully	50	SIGM
		Sal	35	PODS
		Sue	38	PODS
EE	MC	Pat	65	PADS
		Piree	54	IEE
		Sandy	36	IEEE
		Sharon	35	PODS
		Sam	40	PODS
ME	MD	Pat	65	PADS
		Ping	57	MEE
		Sandra	35	MEE
		Syl	37	MDS

Figure 3.11: Staff of the Faculty of Engineering

General Operation

We can also embed general relational expressions into domain algebra. This is called *general operation*. "General" here means more general than the operation we introduced before in this Chapter. However, it is not arbitrarily general. We will show the limitations imposed on it at the end of this Chapter.

In the Faculty of Engineering, rich professors are professors whose yearly salary equals or exceeds 55 K. We have the query: Find the rich engineering professors together with their salary and the department they belong to. The following expression will answer the query:

> **let** *RichProf* **be** "< [*Name, Salary*] **where** *Salary*>=*55* **in** *Professor* >";

> *RP* < − [*Dept, RichProf*] **in** *FactEng*;

> **pr!!** *RP*;

The result is shown in Figure 3.12.

RP		
Dept	*RichProf*	
	Name	*Salary*
CS	Pat	65
	Paul	55
EE	Pat	65
ME	Pat	65
	Ping	57

Figure 3.12: Rich Professors of Engineering Departments

We can make more complicated general operations. For example, we can do **sjoin** on different domain names in two nested domain_relations.

Query: Find professors and secretaries such that the secretary works for all the committees to which the professor belongs.

```
>    let Pname be Name
>    let Sname be Name
>    let ProfSecr be
     "< ( [ Pname, Commit ] in Professor) sub ( [ Sname, Commit ] in Secretary) >"
>    PSC  <- [ Dept, ProfSecr ] in ED
>    pr!! PSC
```

PSC		
Dept	*ProfSecr*	
	Pname	*Sname*
CS	*Paul*	*Sal*
	Paul	*Sue*
EE	*Pirre*	*Sandy*
ME	*Ping*	*Sandra*

Figure 3.13: Professors and Secretary in Committes

The formal syntax:

```
<domain_relational_statement>  :=  'let' <nest_domain_name> 'be'
                                   ' "<' <relational_expression> ' >"'
```

<relational_expression> is an expression of relational algebra operations with some limits. The T-selector in the following paragraph illustrates this. Note that we quote <relational_expression> using “"< >"”, and during declaration, it is treated as string, yet during the actualization, the Relix statement included in the string will be evaluated.

<T-selector> := '[' <attribute_list> ']' 'where'
<selection_clause> 'in' <nested_domain>

<selection_clause> is a comma-separated list of simple logic domain expression that can be evaluated horizontally to true or false on each tuple of the operand <nested_domain> (which is a relation as well).

We have not been able to implement vertical domain operations within the syntax of general operations (in <relational_expression>).

Chapter 4

Implementation of Nested Relations

This chapter deals with the implementation of nested relations. Section 4.1 gives an overview of the implementation of Relix. Section 4.2 describes how nested relations are represented and declared. Section 4.3 illustrates the implementation of nested relation operations.

4.1 Implementation of Relix

Relix is an interactive multi-user system written in C, and is portable across different platforms running the UNIX operating system. Extensions in Relix require that the modules to be added are compatible with the existing code. Therefore, in this section we overview the implementation of Relix that is related to the work of this thesis. A complete documentation for its first implementation can be found in [Lal86].

4.1.1 System Relations

A relation is stored in a UNIX file whose name corresponds to the name of the relation. A database, which is a collections of relations, is equivalent to a UNIX directory. Every Relix database maintains a set of system relations which represents the data dictionary of the database and are stored permanently as UNIX hidden files.[1] Three basic system relations are used to store information about domains and relations in the database.

1. *.rel (.rel_name, .sort_status, .rank, .ntuples)*[2]

   The *.rel* system relation stores information about all the relations in the database.

   - .rel_name is the name of the relation
   - .sort_status specifies the type of sorting for the relation, such as sorted, non-sorted and partly sorted
   - .rank is the number of sorted attributes in the relation
   - .ntuples is the number of tuples in the relation

2. *.dom (.dom_name, .type)*

   The *.dom* system relation stores information about all the domains in the database.

   - .dom_name is the name of the domain
   - .type is the data type of the domain. There are 6 atomic data types (see Figure 2.2)

[1] File names beginning with a period (.) are UNIX hidden files which are not normally listed under the UNIX list directory command.

[2] In Relix convention, the names which begin with a period (.) are system names.

3. *.rd (.rel_name, .dom_name, .dom_pos, .dom_count)*

   The *.rd* system relation stores information that links the relations with the domains on which they are defined.

   - .rel_name is the name of the relation
   - .dom_name is the name of the domain
   - .dom_pos is the byte position of the domain in the relation
   - .dom_count is the number of domains in the relation

In our implementation of nested relations, we use two system relations to store the interface information for the nested relations declared in the database.

1. *.nst (.sup_name, .sub_name)*

   The *.nst* system relation contains information about parent relations and their child relations.

   - .sup_name is the name of the parent relation
   - .sub_name is the name of the child relation

2. *.nest_dom (.domain_name, .domain_ref)*

   The *.nest_dom* system relation contains information about the nested domains.

   - .domain_name is the name of the nested domain (child relation)
   - .domain_ref is the number of reference times of this domain

4.1.2 Parser and Interpreter

Relix consists of two main modules: a parser and an interpreter. The parser, which is generated by Lex [Les75] and Yacc [Joh75],performs syntax analysis and generates intermediate codes. The interpreter is written in C, it reads instructions from the intermediate code and calls particular C functions to perform the operations. Figure 4.1 summarizes the main flow of Relix.

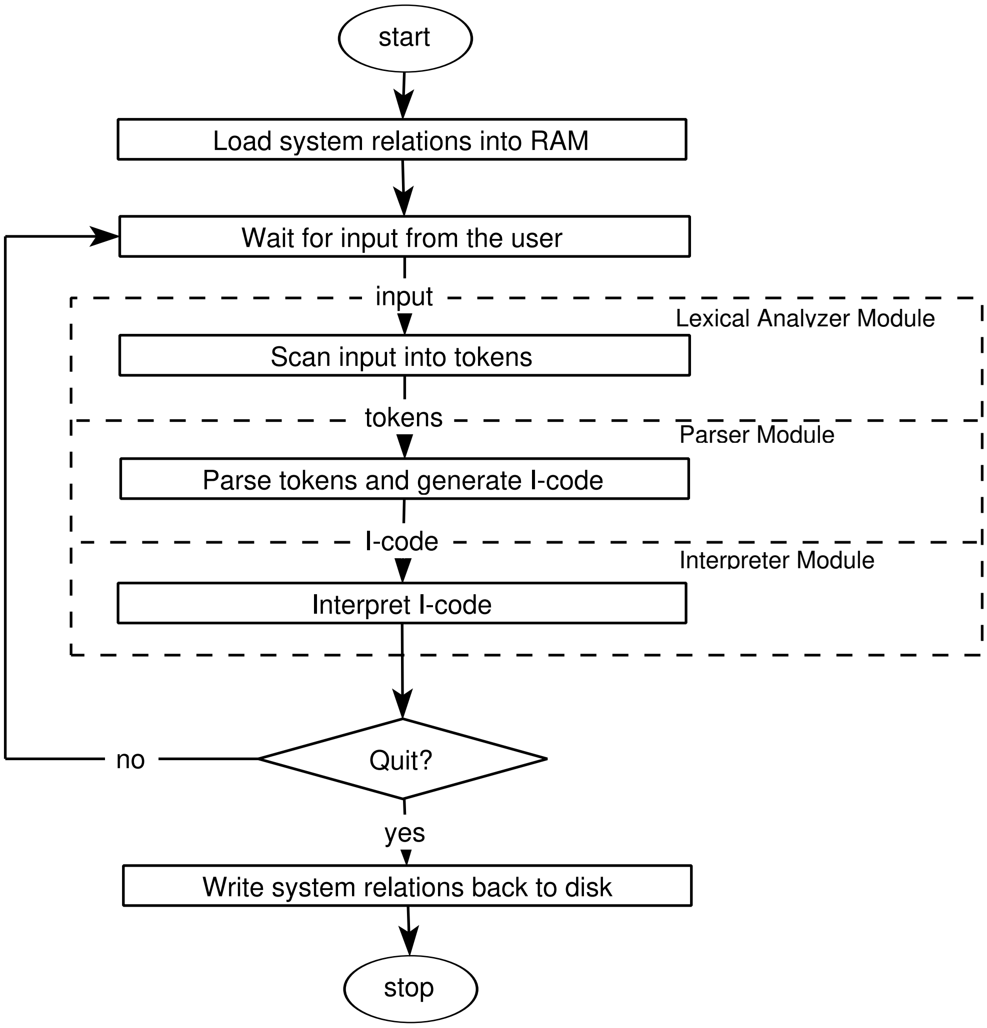


Figure 4.1: Relix Execution Flowchart

We will show an example from an implementation point of view to exemplify how Relix operates.

Suppose we have:

> **domain** a **intg** ;

The parser performs syntax analysis and finds that the above statement fits the following grammar rules.

```
domain_declaration:
     DOMAIN_DEC identifer
       { translator( DOMAIN_DEC);}
     TYPE
       { translator( IDENTIFIER); translator( TYPE); }
     ;
```

Actions in Yacc are C codes enclosed in a pair of curly brackets. The translator function is a C function which performs various tasks according to the actual parameters. The tasks of the translator function include:

- maintaining a scalar stack for storing and retrieving identifiers
- maintaining a set of flags and counters
- generating I-code

For instance, the call 'translator(IDENTIFIER)' pushes the value of the identifier onto the scalar stack.

Some of the parameters produce I-code. For example:

parameter	I-code
DOMAIN_DEC	`global-dom`
TYPE	`push-name a domain`

'a' is a string obtained by popping an item from the scalar stack. The I-code for the example statement is shown below:

```
global-dom          /*set the flag notifying that the following
                      declared domain is a global domain. */
a
push-name           /* Push the next string onto the stack.*/
long
push-name
a
domain              /* Pop 'a' from the stack, and declare
                       'a' as an integer domain. */
halt                /* Update system relations and return. */
```

The comments on the right hand side describe the interpreter actions for the corresponding I-codes. The interpreter maintains a stack for storing and retrieving operands. The 'push-name' pushes an operand onto the stack. The 'domain' is a collection of C functions that the interpreter needs to call with predefined arguments, which are obtained by popping the operands from the stack. Note that '`halt`' is required at the end of the I-code for the interpreter to stop execution.

4.1.3 Implementation of Domain Operations

Suppose we define a virtual domain D as a function of other domains (see Section 2.3). In the implementation, we have routines which will locate these domains in relation R, calculate the corresponding values of D from these operands and append these values of D to the appropraite tuples of the original relation.

The following example will show how domain operations work in Relix:

We declare a constant atrribute as follows:

> **let** *a* **be** *5*;

After the declaration, the domain 'a' is recored in the system as:

```
       Name                Actual  Visited Label  Type
-------------------------------------------------------------
    a                      FALSE   TRUE    1     short
                                Operator:  constant
                                Value:     "+00005"
```

Note that the '*Actual*' value of domain a is *false*, which means that a is a virtual domain, and the following Relix statement requires it to be actualized.

$> \quad ACT \ \ <- [\ a\ ]$ **in** $TEST$;

The I-code for the example statement is shown below.

```
push-name         /* Push the next string onto the stack. */
ACT
name
constant-relation /* Call function constant_relation to
                     create a new relation using the name
                     on the stack */
push-name
TEST
push-name
a
push-count        /* Push a counter onto the stack. */
1
project           /* Call function project to
                    create a new relation according to
                    the attributes required */
assign-scalar     /* Pop item A and B from the stack, and
                    call function assign_scalar to
                    assign item A to item B. */
halt              /* Update system relations and return. */
```

In above I-code, when the intepreter reads **project**, it will call a C function 'project()' to perform the actual projection. In turn, porject() will call yet another function 'actionize_if_any_virtual()' to actualize the virtual domains ('*a*' in this case).

The algorithm for routine *project()* is as follows:

project(list_R, r_name)

where list_R *is a linked list which contains the domains to be projected and* r_name *is the name of the relation on which the domains are to be projected.*

1. *Check list_R, make sure no duplicates are included.*
2. *Actualize list_R from r_name to R (a temporary file). Sort R on list_R. Call the routine actualize_if_any().*
3. *Do actual projection according to list_R.*
4. *Return the file name of the results of projection.*

The algorithm for routine *actualize_if_any_virtual()* is :

actualize_if_any_virtual(R_name, E_list)

where R_name *is the name of the relation being processed and* E_list *is a list of attributes of the relation in* R_name*, including both the original attributes and virtual attributes which are defined as a function of the original attributes.*

1. *Traverse the attribute list and find if there are any virtual domains.*
2. *If there are no virtual domains, return the original relation.*
3. *If there exist virtual domains.*
   (a) *Traverse each tuple of the orginal relation.*
   (b) *Actualize the virtual domain value according to the definition of the virtual domain.*

   (c) *Put all the tuples in a temporary relation.*

   (d) *Return the temporary relation.*

In our example, the program flow is as follows:

1. *When* project() *is called, the values in the two parameters are:*

   (a) list_R, *which points to a list which includes only one item, 'a'.*

   (b) r_name, *which is 'TEST'.*

2. *Then actualize_if_any() is called with the parameters' values as:*

   (a) E_list, *which points to a list which is the same as* list_R *in project(), i.e., 'a'.*

   (b) r_name, *which is the same as* r_name *in project(), i.e., 'TEST'.*

3. *In actualize_if_any(), the sytem finds that 'a' is a virtual attribute, and thereafter, domain* a *is actualized by assigning the value of 5 to the attribute 'a' of every tuple in* TEST.

4. *Actualize_if_any() returns the name of the temporary relation to project(), which in turn projects the* 'a' *domain and returns the result to system.*

5. *Update system tables.*

4.2 Declaration and Initialization of Nested Relations

4.2.1 Declaration of Relation Data Type

We can declare a regular integer domain S and a regular relation S with domains a and b as follows:

> **domain** S **intg** ;

> **relation** S (a , b);

We have already explained the I-codes of domain declaration (see Section 4.1.2). The I-codes of the relation declaration is as follows:

```
push-name
no-cp-ln          /* Set the flag that only declares. There is
                     no data input*/
push-name         /* Push the next string onto the stack.*/

push-name
a
push-name
b
push-count
2                 /* number of domains */
push-name
S
relation          /* Pop domain list (a and b) from the stack,
                     pop S from the stack, and declare S as a
                     relation */
halt              /* Update system relations and return. */
```

To declare a relation data type, we combine the above two cases and add the following grammar to yacc:

<nested_domain_declaration> := 'domain' <identifier> <domain_list>

For instance:

$>$ **domain** S (a , b);

The resulting I-codes are also a combination of the above:

```
push-name
no-cp-ln
push-name

push-name
.id                /* Add a system domain .id to refer to
                      the parent relation */
push-name
a
push-name
b
push-count
3
relation
global-dom
S
push-name
relation
push-name
S
domain
end-dom-code
halt
```

The comparison of the above three cases is shown in Figure 4.2.

domain S intg;	*relation S (a,b);*	*domain S (a,b);*
global-dom	push-name	push-name
S	no-cp-ln	no-cp-ln
push-name	push-name	push-name
long		
push-name	push-name	push-name
S	a	.id
domain	push-name	push-name
end-dom-code	b	a
	push-count	push-name
	2	b
	push-name	push-count
	S	3
	relation	relation
		global-dom
		S
		push-name
		relation
		push-name
		S
		domain
		end-dom-code

Figure 4.2: Comparison

Each nested domain has its declaration entry in both *.dom* system table and *.rel* system table. The *.type* in table *.dom* of any nested_domain, i.e., relation data type, is set to a constant 'RELATION', which equals 11 in the current version. The following entry in *.dom* table is for the nested domain *S*:

.dom	(.dom_name,	.type)
	S	11

The following entry in *.rel* table is also for the nested domain *S*:

.rel	(.rel_name,	.sort_status,	.rank,	.ntuples)
	S	0	0	0

Because nested domain S is a relation itself, its information and that of its domains are stored in another system table *.rd*. The following entry is for *S*:

.rd	(.rel_name,	.dom_name,	.dom_pos,	.dom_count)
	S	.id	0	-3
	S	a	1	-3
	S	b	2	-3

Note that *S* has three domains, among which *.id* is added by the system in order to make reference to the parent relation.

S also has an entry in the system table *.nest_dom*.

.nest_dom	(.domain_name,	.domain_ref)
	S	0

4.2.2 Initialization

Initialization of relations can be achieved by supplying the initialization data directly on the command line:

> **relation** *Simple* (*a,b*) <- {(*1,2*),(*3,4*)};

For flat relations, the algorithm of initialization is:

1. *Parse the relation identifier and parse the domain identifiers. In the above case, 'Sample', 'a', and 'b', then create a file named* 'Simple'*.*

2. *Parse the constants, and save the constants to file* 'Simple'*.*

Recall that we declare the nested domain:

> **domain** S (a , b);

For nested relations, we can initialize as follows:

> **relation** *TEST* (*c*, *S*) <- {(*3*,{(*1*,*2*),(*8*,*7*)}),(*7*,{(*6*,*5*),(*4*,*9*)})};

since we include a nested domain S here, we need to revise the algorithm to achieve the desired effects.

1. *Parse the relation identifier and the domain identifiers, and record the nested subrelations (nested domains). Then create a file named 'Test', also create files according to subrelations, in this case we have* 'S'*.*

2. *Parse the constants. When we meet a curly brace '{', we create a surrogate to the parent attribute, and put the corresponding real constants into the corresponding subrelations. For example, for {(1,2),(8,7)}, the surrogate is 0 and for {(6,5),(4,9)}, the surrogate is 1. Thus,*

   (a) *In file* TEST*, we have (3,0), (7,1);*

   (b) *In file* S*, we have (0,1,2),(0,8,7),(1,6,5),(1,4,9);*

4.3 Operations

In this section, we present the implementation for operations on nested child relations (nested domains).

4.3.1 Implementation of Reduction

We will show by example how reduction operates on nested relations in Relix. Since we based our implementation on the existing implementation of reduction on scalar attributes, we will first present the implementation of reduction on scalar attributes.

Reduction on scalar attributes

Scalar attributes' data types are atomic as summarized in Figure 2.2. Recall that in Chapter 2, we already listed that what scalar operations can be conducted on both simple reductions and equivlant reduction. Now we will show how they are implemented by using an example of '+', the add operator.

Suppose we have a database *order* as in Figure 4.3.

```
Order
-----------------------------------------
Customer        Product         Amount
-----------------------------------------
Ann             W               10
Ann             X               40
Ping            M               20
Sam             Y               30
```

Figure 4.3: Order table

In order to gain the total order Amount of all the customers, we can use our 'red +' operator, and impose it on the domain *Amount*.

> **let** *Total* **be red + of** *Amount* ;

Domain *Total* is kept in the system as:

```
     Name              Actual  Visited Label  Type
------------------------------------------------------
     Total              FALSE   TRUE   51      long
                        Operator:  red-plus
                        Operand-1: Amount
```

Whenever a Relix statement wants to include *Total*, the system will call *Actualize_if_any()* to actualize it.

As we can see, *Total* is defined on *Amount*.

The algorithm is as follows:

1. *Initialize an accumulator according to* Amount *(In this case, its data type is* long*).*

2. *Scan through each tuple of the relation* Order. *Extract the value of* Amount, *add it to the accumulator (Recall that operator of* Total *is '+').*

3. *Assign the value in the accumulator to the* Total *attribute of each tuple.*

Thus we can actualize *Total* and the result is shown in Figure 4.4.

```
Order
- - - - - - - - - - - - - - - - - - - - - - - -
Customer       Product        Amount         (Total)
- - - - - - - - - - - - - - - - - - - - - - - -
Ann            W              10             100
Ann            X              40             100
Ping           M              20             100
Sam            Y              30             100
```

Figure 4.4: Values of Total after actualization

Furthermore, we would like to know the total amount of the products each customer ordered. The follwing Relix statement can help us to perform this task:

> $>$ **let** *CusTotal* **be equiv + of** *Amount* **by** *Customer* ;

It is stored in the system as:

```
         Name          Actual   Visited Label   Type
-------------------------------------------------------
       CusTotal         FALSE    TRUE      52      long
                        Operator:  equiv-plus
                        Operand-1: Amount
                        By-list:   Customer
```

We can see in the system data structure that *CusTotal* actually has an item called *by-list*, which includes *Customer*, and that the resulting *CusTotal* will be based on this list.

With following steps we can actualize *CusTotal*:

1. *Sort original relation* Order *on* by-list *(i.e., 'Customer').*

2. *Initialize an accumulator storage according to* CusTotal

3. *Scan through tuples of* Order*, if the tuple's value is kept the same in attribute* Customer*, add it to the accumulator, otherwise append the value of the accumulator to the previous tuples, and reset the accumulator.*

This way we can actualize *CusTotal* as shown in Figure 4.5.

Reduction on Nested Attributes

In this section, we will present the general algorithms of reduction on nested attributes first and then show some examples.

The operator of reductions on nested attributes falls in one of the following groups:

```
Order
- - - - - - - - - - - - - - - - - - - - - - - -
Customer       Product        Amount         (CusTotal)
- - - - - - - - - - - - - - - - - - - - - - - -
Ann            W              10             50
Ann            X              40             50
Ping           M              20             20
Sam            Y              30             30
```

Figure 4.5: Value of CusTotal after actualization

(simple_reduction	equivalence_reduction)
red_ijoin	equiv_ijoin
red_ujoin	equiv_ujoin
red_sjoin	equiv_sjoin

General Algorithm

- Simple Reduction

  In this case, the operator belongs to the first group.

  1. *In the parent relation level, we assign each tuple in the position of the operand domain a constant 0. For simple reduction, the value of this attribute should have the same value for all tuples in the relation.*
  2. *In the nested relation level, according to the operator, do* ujoin, ijoin *and* sjoin *with the subrelations (which are actually stored in the same physical table).*
     (a) ujoin*: Project all the attributes except* .id*. The obtained result is the required ujoin operations on those sub-relations. Then, append a new .id to it, in order to keep links with the parental relation. The value is a constant* 0.

(b) ijoin*: Sort the table according to the number of tuples in each sub-relation, select the sub-relations one by one according to the value of* .id *and do* ijoin *on them. In this way, we can improve the join efficiency, since during the join procedure, the result might be empty before we reach the last subrelation.*

(c) sjoin*: The algorithm is the same as* ijoin*, except we do not need to sort the table.*

- Equivalence Reduction

  In this case, the operator belongs to the second group.

  1. *Sort the original relation on by_list.*
  2. *Determine equivalence classes, for each class, do* inside reduction*, which will be presented next.*

  Inside Reduction

  1. *Initialize an accumulator, which is an empty temporary relation.*
  2. *For each tuple:*
     Extract the value of the nested domain, i.e., the pointer to the underlying subrelation;
     Extract tuples of subrelation according to the mapping between the parent nested domain and .id, store them in a temporary file.
     Perform the appropriate join (ijoin, ujoin, sjoin) with the accumulator.

Examples

In Figure 4.6, we have a relation *Order_book* with domains *Customer* and *Order*, which is a subrelation with domain *Product*.

Order_Book

Customer	Order
Ann	0
Ann	1
Ping	2
Sam	3

Order

.id	Product
0	W
0	X
1	W
2	M
2	W
3	W
3	Y

Figure 4.6: Relation Order_Book and its subrelation Order

We have three Relix statements:

1. > **let** *AllProduct* **be red ujoin of** *Order* ;
2. > **let** *IProduct* **be red ijoin of** *Order* ;
3. > **let** *CustProduct* **be equiv ijoin of** *Order* **by** *Customer* ;

The first Relix statement above finds all the products ordered by the customers. The second one finds products which are ordered in each individual order. The third one finds all the products ordered in every order by each customer.

To actualize *AllProduct* , we can run the Relix statement:

> *Order_Book1* $<-$ [*Customer, AllProduct*] **in** *Order_Book* ;

System running flow:

1. *Operator* red ujoin *belongs to the first group*

2. *In* Order_Book*, we assign* AllProduct *a constant* 0

3. *In the nested relation level, i.e.,* AllProduct*, the operator is* red ujoin *and the operand is* Order*. We project [Product] from* Order*, and append a new .id to*

each tuple of the new obtained relation, in order to keep links with AllProduct *in* Order_Book. *Thus we have a new subrelation* AllProduct.

4. *Update system tables.*

The actualized *AllProduct* is shown in Figure 4.7.

Order_Book1

Customer	AllProduct
Ann	0
Ping	0
Sam	0

AllProduct

.id	Product
0	M
0	W
0	X
0	Y

AllProduct: red ujoin of Order

Figure 4.7: AllProduct in relation Order_book1

To actualize *IProduct* , we can run the Relix statement:

$> \quad \textit{Order\_Book2} \;\; < - \; [\textit{Customer, IProduct}\,]$ **in** *Order_Book* ;

System running flow:

1. *Operator* red ijoin *belongs to the first group*

2. *In* Order_Book, *we assign to* IProduct *a constant* 0

3. *In the nested relation level (i.e.,* IProduct*) the operator is* red ijoin *and the operand is* Order. *We do ijoin between the different set of Product values according to .id. They are* {*(W), (X)*}, {*(W)*}, {*(M),(W)*} *and* {*(Y),(W)*} *respectively. The result is* {*(W)*}. *In order to keep links with IProduct in Order_Book,*

we append a new .id to each tuple of the new obtained relation. Thus we have a new subrelation IProduct.

4. *Update system tables.*

The actualize *IProduct* is shown in Figure 4.8.

```
Order_Book2                                 IProduct
 - - - - - - - - - - -                      - - - - - - - - -
Customer  AllProduct                        .id     Product
 - - - - - - - - - - -                      - - - - - - - - -
Ann              0 )                        0       W
Ping             0 )- - - - - - - - - - - -  - - - - - - - - -
Sam              0 )
 - - - - - - - - - - -
```

IProduct: red ijoin of Order

Figure 4.8: IProduct in relation Order_book2

To actualize *CustProduct*, the following Relix statement can satisfy the requirement:

$>$ *Order_Book3* $<-$ [*Customer, CustProduct*] **in** *Order_Book* ;

System running flow:

1. *Operator* equiv ijoin *belongs to the second group*

2. *Sort* Order_book *on* Customer

3. *For each* Customer*: determine equivalence classes, and conduct* ijoin *within each class. For example, for customer* Ann*, we first extract* $\{(W),(X)\}$*, then* $\{(W)\}$*. After doing ijoin between them, we get* $\{(W)\}$*;*

4. *Update system tables.*

The actualized *CustProduct* is shown in Figure 4.9.

```
Order_Book2                         CustProduct
------------                        ----------
Customer CustProduct                .id   Product
------------                        ----------
Ann             0 ------ 0            W
Ping            1 ------ 1            M
Sam             2 ------ 2            W
------------      ------ 2            Y
                                    ----------
```

CustProduct: equiv ujoin of Order

Figure 4.9: CustProduct in relation Order_book3

4.3.2 Horizontal Operation

Binary Operation

The operators of binary operation are: *ujoin*, *ijoin*, and *sjoin*.

General Algorithm

1. *In the parent relation level, copy the value from one of the operands' to the new domain.*

2. *In the subrelation level, call Relix again to obtain the new subrelation.*

3. *Join back the obtained subrelation to the parent relation on subrelation's* .id *attribute with parental relation's attribute.*

4. *Update system table.*

Example

In Figure 4.10, we have relation *Order_Book* with domains *OldOrd*, *Customer* and *NewOrd*. *OldOrd* and *NewOld* are nested domains.

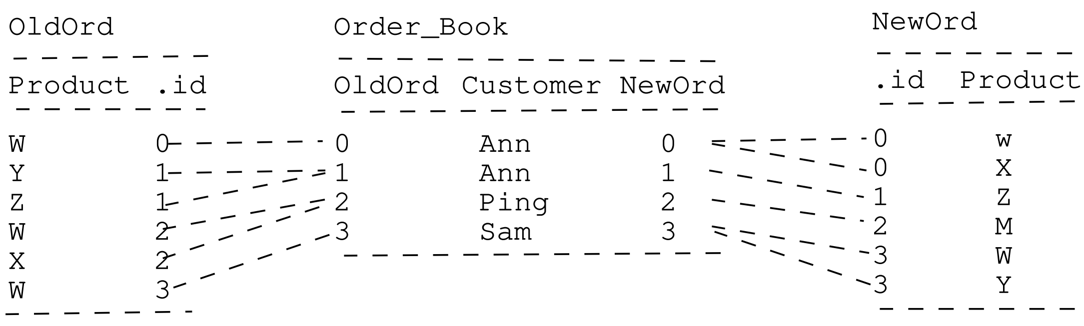


Figure 4.10: Relation Order_Book with subrelations OldOrd and NewOld

Suppose we have:

> **let** *Order* **be** *OldOrd* **ujoin** *NewOld* ;

and we can actualize Order using the following statement:

> *Order_Book4* $<-$ [*Customer, Order*] **in** *Order_Book* ;

The procedure of actualizing Order:

1. *Copy* OldOrd *to* Order. *This way, we can keep a set of surrogates of* Order *in parent relation* Order_Book.

2. *Call Relix again to get* Order, *i.e., run "Order* $<-$ *OldOrd ujoin NewOrd" in Relix. Since both* OldOrd *and* NewOrd *have same attributes, .id and Product, we do ujoin on them to get* Order.

3. *Join back the obtained subrelation to the parent relation on subrelation's* .id *attribute with the parent relation's attribute* Order. *"Order_Book* $<-$ *Order_Book*

[Order ijoin .id] Order"

The final result is shown in Figure 4.11.

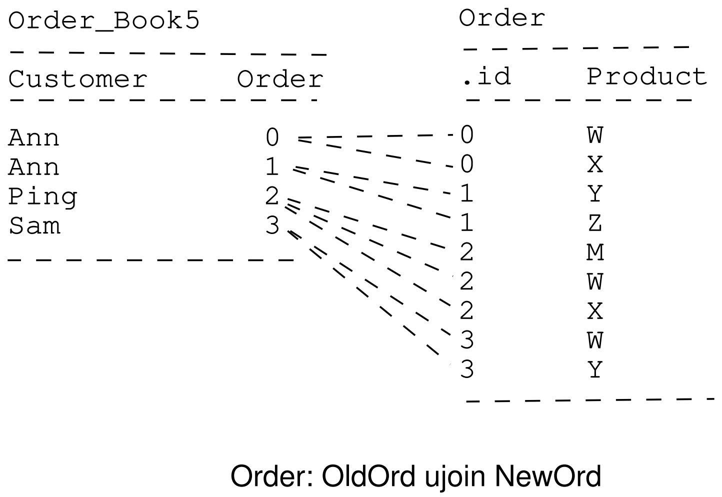


Figure 4.11: Actualized result of Order in relation Order_Book

General Operation

General Operations are stored as strings when they are declared. Suppose we have the relation as shown in Figure 4.12 and the following query:

> **let** *BigOrd* **be** "< [*Product*] **where** *Amount* > *8* **in** *Order* >";

Domain *BigOrd* is stored as:

```
Name           Actual  Visited Label Type
-------------------------------------------------------------
BigOrd         FALSE   TRUE    52   relation
               Operator:  t-dom
               Operand: [Product] where Amount > 8 in Order
```

Order_Book

Customer	Order
Ann	0
Ann	1
Ping	2
Sam	3

Order

.id	Product	Amount
0	W	9
0	X	6
1	Z	10
2	M	12
3	Y	10
3	W	7

Figure 4.12: Relation Order_Book

And the following statement will actualize *BigOrd*:

$>$ *Order_Book5* $<-$ [*Customer, BigOrder*] **in** *Order_Book* ;

The procedure of actualizing *BigOrd* is as follows:

1. *In the parent level, copy* Order *to* BigOrd.
2. *Extract the relational statement from the string, parse it (the parser will be described in next section); the string will be altered from "[Product] where Amount* $>$ *8 in Order" to "[.id, Product] where Amount* $>$ *8 in Order".*
3. *Call Relix to get the resulting subrelation, "BigOrd* $<-$ *[.id, Product,] where Amount* $>$ *8 in Order".*
4. *Join back the resulting subrelation with the parent relation on .id. "Order_Book* $<-$ *Order_Book [BigOrd ijoin .id] BigOrd".*
5. *Update system tables.*

The result is shown in Figure 4.13

```
Order_Book                       BigOrd
-----------                      ------------
Customer   BigOrd                .id  Product Amount
-----------                      ------------
Ann           0  ------- 0        W       9
Ann           1  ------- 1        Z       10
Ping          2  ------- 2        M       12
Sam           3  ------- 3        Y       10
-----------                      ------------
```

Figure 4.13: Actualized BigOrd

Parser

In general domain algebra operations, we can write regular relational expressions with some limitations, i.e., we can not include vertical operations in the quoted relational expression.

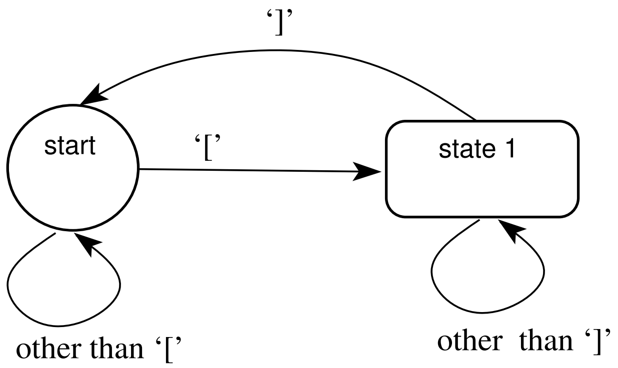
Since we call Relix again to get the resulting relation, we need to preprocess the statement. We build a small parser to preprocess the expression.

For example, '*[Product] where Amount > 8 in Order*' will become '*[.id, Product] where Amount > 8 in Order*'. The automaton of the parser is shown in Figure 4.14. Suppose we have "A [a ijoin b] B". The flow of its automaton is:

1. The automaton reads 'A'. It stays at the start. The output is "A".
2. The automaton reads '['. It goes to state 1. The outputs is "A [".
3. The automaton reads 'a'. It stays at state 1. The output is "A [a"
4. The automaton reads 'ijoin'. It stays at state 1. The output is "A [a, .id ijoin"
5. The automaton reads 'b'. It stays at state 1. The output is "A [a, .id ijon b"
6. The automaton reads ']'. It goes back to the start. The output is "A [a, .id ijoin b, .id]"

7. The automaton reads ‘B’. It stays at the start. The output is “A [a, .id ijoin b, .id] B”

8. The automaton reads EOF. It stops and returns the obtained output.



Algorithm:

```
For state start:
 if next token is '[', go to state 1, else stay at state start

For state 1:
 if  next token is "]"
        add .id before "]", i.e. ".id ]"
        go to state start
    else if next token is any join token
           add .id before the join token, for example, ".id ijoin"
        stay at state 1


join token:    ijoin, djoin, ujoin, sjoin, ljoin, rjoin,
               drjoin, natjoin dljoin, natjoin, dljoin,
               gtjoin, sup, eqjoin, sub, ltjoin, sep,
               qejoin , lejoin, iejoin, div, ~gejoin, ~sup,
               ~eqjoin, ~sub,  ~ltjoin, icomp, natcomp
```

Figure 4.14: The parser to parse the embedded general relational expression

Chapter 5

Conclusion

Nested relations have been explored thoroughly in past decades, with the major research direction focused on nesting and unnesting [Jae82][Fis85][Kor89][Tak89]. In our approach, we build nested relations upon flat relations. We show that flat relations are powerful enough to model nested relations and to facilitate nested relation queries. The purpose of this thesis is to begin to integrate nested relations into a relational database programming language (Relix)by integrating the relational algebra into the domain algebra.

5.1 Summary

We built our nested relation model upon the original Relix database model. Relix is powerful enough to support nested relations. No modifications have been made to the original database engine itself. However some extensions were made to facilitate the process of integration and to provide new features.

- A new system attribute *.id* has been added to Relix , which provides a way of linking the parent relation to its included nested relations.

- One level of nesting has been integrated into Relix.

- A part of the relational operator can be added to the domain algebra. This partially eliminates the difference between domains and relations.

Our implementation showed that Relix is powerful enough to include nested relations, and that it is convenient to add nested relations to the system. The relational operations, such as *ujoin*, *sjoin*, *ijoin*, which are added to domain operations, function well.

However, the surrogate mechanism we used is a bit simple, and we have not been able to include more information in the surrogates except to use it to keep links between nested child relations and the parent relation. No large-scale tests have been done, since it is beyond the scope of this M.Sc. thesis.

5.2 Future Work

So far, we have only implemented one level of nesting in Relix, which is the first step towards fully implementing the features of nested relations. There are still more features that can be added such as:

- Implementing multiple nesting and recursive nesting. To date, we have only implemented one level of nesting, which provides a prototype for multiple nesting. Theoretically, it is possible to build infinite levels of nested relations.

- Fully integrating the relational algebra into the domain algebra. Only a part of relational algebra has been integrated into domain algebra to date. Further work can be done on functional mapping and partial function mapping on nested relations.

- Combining nested relations with procedure abstraction and to implement complex objects. A procedure facility has been recently added to the Relix system [Lui96]. We could extend certain procedures to nested relations. Those procedures can be viewed as methods to manipulate a certain nested relation, which can then be treated as a complex object.

Bibliography

[Cod70] E. F. Codd. A Relational Model of Data for Large Shared Data Banks. *Communications of the ACM*, 13(6), Oct. 1970, pp.337-387

[Cod72] E. F Codd. A Data Base Sublanguage Founded on the Relational Calculus. Proceedings of 1971 ACM SIGFIDET Workshop on *Data Description, Access and Control.*

[Des88] A. Deshpande, D. Van Gucht. An implementation for Nested Relational Database. Proceedings of the 14th International Conference on *Very Large Data Bases*, April 1988, pp. 266-274

[Fis85] P. C. Fischer, D. Van Gucht. Determining when a Structure is a Nested Relation. Proceedings of the 11th International Conference on *Very Large Data Baes*, August 1985, pp. 171-180

[Jae82] G. Jaeschke, H-J. Schek. Remarks on the Algebra of Non-First-Normal-Form Relations. *Proceedings of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems.* March 1982, pp.124-138

[Joh75] S. C. Johnson. Yacc: Yet another compiler-compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Kor89] H. F. Korth, M. A. Roth. Query Languages for Nested Relational Databases. Nested Relations and Complex Objects in Database. *Lecture Notes in Computer Science*, Springer-Verlag, New York 1989.

[Lal86] N. Laliberté. Design and Implementation of a Primary Memory Version of Aldat. Master's thesis, McGill University, Montreal, Canada, 1986.

[Les75] M. E. Lesk. Lex: a lexical analyzer generator. Technical Report 39, AT&T Bell Laboratories, Murray Hill, N.J., 1975.

[Lev92] M. Levene. The Nested Universal Relational Database Model. *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1992

[Lui96] R. Lui. Implementation of Procedure in a Database Programming Language. Master's thesis, McGill University, Montreal, Canada, 1996.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. *Proceedings of 3rd International Conference on VLDB*, Tokyo, pp. 447-453, 1977.

[Mer76] T. H. Merrett. MRDS: An Algebraic Relational Database System. In *Canadian Computer Conference*, Montreal, pp.102-124, May 1976

[Mer77] T. H. Merrett. Relations as programming language elements. *Information Processing Letters*, 6(1):29–33, Feb. 1977.

[Mer84] T. H. Merrett. *Relational Information Systems.* Reston Publishing Company, Reston, Virginia, 1984.

[OOM87] G. Ozsoyoglu, Z. M. Ozsoyoglu, V. Matos. Extending relational algebra and relational calculus with set-valued attributes and aggregate functions. ACM Transaction on Database Systems, 12(4) Dec. 1987, pp. 566-593

[Ozy87] Z. M. Ozsoyoglu & L. Y Yuan. A design method for nested relational databases. Proceedings of 3rd IEEE conference on Data Engineering, Los Angeles, pp. 599-608, 1987

[Ozy89] Z. M. Ozsoyoglu & L. Y Yuan. On Normalization in Nested Relatonal Databases. Nested Relations and Complex Objects in Database. *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1989.

[Pis86] P. Pistor, F. Anderson. Designing a Generalized NF^2 Model With An SQL_Type language Interface. Proceedings of the 12th International Conference on Very Large Data Bases, August 1986, pp. 278-285.

[Pvg92] J. Paredaens, D. Van Gucht. Converting Nested Algebra Expressions into Flat Algebra Expressions. ACM *Transactions on Database Systems* 17(1), March 1992, pp. 65-93.

[RKS86] M. A. Roth, H. F. Korth, A. Silberschatz. Extended algebra and calculus for nested relational databases. ACM *Transactions on Database Systems* 13(4), Dec. 1988, pp. 390-417.

[Sch82] H. J. Schek, P. Pistor. Data Structure for an Integrated Data Base Management and Information Retrieve System. Proceedings of the 8th International Conference on *Very Large Data Bases*, Sep. 1982, pp. 197-207.

[Sps87] M. H. Scholl, H. B. Paul, H. J Scholl. Supporting Flat Relations by a Nested Relational Kernel. Proceedings of the 13th International Conference on *Very Large Data Bases*, Sep. 1987, pp. 137-147.

[Sab89] 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 Database, *Lecture Notes in Computer Science*, Springer-Verlag, NY, 1989.

[Stone96] M. Stonebraker. *Object-Relational DBMSs*. Morgan Kaufmann Publishers Inc., San Francisco, California, 1996.

[Tak89] K. Takeda. On the Uniqueness of Nested Relations. Nested Relations and Complex Objects in Databases, *Lecture notes in Computer Science*, Springer-Verlag, New York, 1989.

[Tag92] A. U. Tansel, L. Garnett. On Roth, Korth, and Silberschatz's Extended Algebra and Calculus for Nested Relational Databases. ACM *Transactions on Database Systems*, 17(2), June 1992, pp. 374-383.

[Tho86] S. Thomas, P. Fischer. Nested relational structures. In Advances in Computing Research III, *The Theory of Databases*, P.C.Kanellakis, Ed. JAI Press, Greenwich, Conn., 1986.