Implementation of The Nested Relational Algebra in Java

Biao Hao

School of Computer Science, McGill University
Montréal, Québec, Canada

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

Contents

List of Figures

List of Tables

Abstract

This thesis discusses the design and implementation of a database programming language in the Java programming language. The language supports relational algebra for nested relations with arbitrary but finite level of nesting, and domain algebra for nested attributes at any level of the nesting hierarchy. Views including recursive view are also implemented.

The support for nested relations is accomplished through four mechanisms. The first is the introduction of new syntax for the declaration of nested attributes and relations, the initialization of nested relations, and the nested update. The second is the integration of relational expressions into domain expressions so that relational operations can be applied to nested attributes. The third is the definition of order between relations so that relation-valued attributes are treated the same way as attributes of primitive type such as integer and string. The forth is to use a set of flat relations which are linked to each other by surrogates, to model a nested relation. With these mechanisms, our implementation provides uniform syntax and notation for relational and domain algebra of nested relations.

The implementation is totally in Java. Therefore it runs on any platform that supports the Java runtime environment.

Acknowledgments

I wish to thank my thesis supervisor Professor Tim Merrett for his guidance, advice, and encouragement throughout the research and preparation of this thesis. He always provides insights into the implementation and the interpretation of testing results. This thesis benefited from his careful reading and constructive criticism.

Thanks to Patrick Baker and Zhongxia Yuan in our project team for the collaboration in the redesign and implementation of Relix in Java. I benefited a lot from the formal and informal discussions with them. Zhongxia's addict to performance and Patrick's thirsty for formality led me to think deeply about our implementation. Patrick helped me write the fancy recursive method for parsing the initialization tree. Zhongxia kindly wrote all the code for dealing with the system relations.

I wish to thank the School of Computer Science for the graduate courses and the research environment. Thanks to Professor Luc Devroye for his wonderful course and discussions. Thanks to Franca Cianci, Vicki Keirl, Josie Vallelonga, Lise Minogue, and Lucy St-James, for easing the procedure of dealing with the school.

I wish to thank my colleagues in the Aldat lab, especially Xiaoyan Zhao for supporting the access to the facilities in the lab and answering many questions about Relix.

Thanks also to the School of Computer Science, Professor Tim Merrett, the Max Stern Recruitment Fellowship, Fonds pour la Formation de Chercheurs et l'Aide à la Recherche (FCAR), and the J. W. McConnell McGill Major Fellowship for financial support.

Finally, I wish to thank my wife Ling Tao, for her support, encouragement during my study.

Chapter 1

Introduction

This thesis describes the design and implementation of a database programming language in the Java programming language [AG96, GJS96]. The basic aim of this implementation is to support relational algebra and domain algebra for relations with arbitrary level of nesting. In section 1.1, we give some background material and motivation for the implementation. In section 1.2, we briefly outline the structure of this thesis.

1.1 Background and Motivation of the Thesis

Since Codd's seminal paper [Cod70] on the relational data model [Cod70, Cod79], the majority of research in database field has been dedicated to some aspect of this model. A solid theoretical foundation has been established for the relational data model. It is also widely accepted in the database application world, and there are many commercial relational database management system packages.

In the original relational model, relations are required to obey the so called first-normal-form (1NF) which means that the values of all attributes of a tuple must be atomic, i.e., undecomposable as far as the relational model is concerned. The 1NF requirement yields flat tabular representations of relations. The relations that are in 1NF are also called flat relations.

The nested relational model [Mak77, SS87, AFS89] was developed in order to extend the flat relational model to applications in which attribute values may have complex structure. The nested relational model removes the restriction of 1NF from the flat relational model, and allows attribute values to be relation-valued as well as atomic. The resulting non-first-normal-form (NF^2) relations are called nested relations. Nested relations provide for the explicit representation of the semantics of the application, and have been used in applications such as office automation, computer aided design, spatial data, expert systems, and geographical information systems.

Relix, as an experimental relational database system, was developed at McGill University [Lal86, MLG^+88]. It provides an interactive environment for exploring the concept of the relational database model. It supports the relational algebra as well as domain algebra for flat relations. Other functionalities such as procedure, event handling, concurrent access, and computation, have been also added to the system. Recently, domain algebra for relations with one-level of nesting has also been implemented [He97].

The purpose of this thesis is to redesign the whole Relix and integrate the notion of nested relations into both the relational algebra and the domain algebra into Relix. Computation is also redesigned to support the notation of objects such as states, local variables, and encapsulated methods. In our implementation, we use the syntax of former Relix wherever it is possible. The support for nested relations is accomplished through four mechanisms. The first is the introduction of new syntax for the declaration of nested attributes and relations, the initialization of nested relations, and the nested update. The second is the integration of relational expressions into domain expressions so that relational operations can be applied to nested attributes. The third is the definition of order between relations so that relation-valued attributes are treated the same way as attributes of primitive type. The forth is to use a set of flat relations which are linked to each other by surrogates, to model a nested relation. With these mechanisms, our implementation provides a uniform syntax and notation for both flat and nested relations, and supports relational algebra for nested relations with arbitrary level of nesting, and domain algebra for nested attributes at any level of the nesting hierarchy.

The new object-oriented programming language Java[1] [AG96, GJS96] is used to implement the system. The advantages of using Java are due to its platform independence[2] and the support of object-oriented features. The program can run on any platform whichever has the Java runtime environment (JRE). The support for object-oriented features and a rich set of application program interface (API) packages make programming easier, especially for the front end of our interactive system. For example, the store and retrieval of the code for virtual domains, views, and computations are greatly simplified using the object serialization mechanism in Java. The big disadvantage is Java's low performance.

Our implementation achieved platform independence for both the program code and the binary data files for relations: the code runs successfully on many platforms including Window 95, Window NT, Linux, and Solaris; the underlying binary data files for databases are reused across these platforms.

1.2 Outline of the Thesis

This thesis discusses the redesign and implementation of former Relix in the Java programming language, particularly the facilities for supporting the relational and domain operations on nested relations.

In chapter 2, we give the background and preliminary material needed throughout the thesis. In section 2.1 we present the flat relational model, in which data is represented in a simple tabular fashion. The former Relix which implements the relational algebra and the domain algebra for flat relations, is discussed. In section 2.2

[1] Java is developed and maintained by Sun Microsystems Inc. It is claimed to be simple, object-oriented, interpreted, architecture neutral and portable, dynamic, multi-threaded, and secure. For detailed information about Java, see Sun's official site at http://www.javasoft.com/.

[2] Both the code and the relation data files of the database are platform independent.

we present the nested relational model, which extends the flat relational model in order to allow the representation of hierarchically structured data. In section 2.3 we describe the approach used in our implementation to support relational algebra and domain algebra for relations with arbitrary level of nesting.

In chapter 3, we describe how to use relational algebra and domain algebra in our implementation. The basic syntax and semantics in our implementation follow closely those of former Relix, with additional syntax added to accommodate nested relations. In section 3.1, we show how to start the interactive interpreter. In section 3.2, we describe the lexical conventions of the interactive language. In section 3.4, we describe the declaration of attributes and relations, and the way to initialize relations at declaration time. In section 3.3, the usage of interactive commands is summarized. In section 3.5, we describe the relational algebra, including selection, projection, join, and update. In section 3.6, we present the domain algebra operations. In section 3.7, we present the use of views. Example queries are used throughout the chapter to show the syntax and semantics of our system.

In chapter 4 we discuss the implementation of the nested relational algebra using Java, with particular emphasis on the way the nested relations are integrated into the frame of flat ones. In section 4.1, we give an overview of the system architecture. In section 4.2, we introduce a way to define order among relations. Once the order is defined, the relation-valued attributes can be treated the same way as the simple attributes in many of the operations on a relation, such as sort and project. The sort algorithm of nested relations, and the storage format of nested relations are presented. In section 4.3, we discuss the implementation of relational expressions, including projection, selection, μ-join and the combination of them. In section 4.4, we describe the implementation of relational statements, including declaration and initialization of relations, assignment, update, and view. In section 4.5, we present some performance results of our implementation.

In chapter 5 we give a brief summary of the thesis and the work to do in the future.

Chapter 2

The Nested Relational Model

In this chapter, we give the background and preliminary material needed throughout the thesis. In section 2.1 we present the flat relational model, in which data is represented in a simple tabular fashion. The former Relix which implements the relational algebra and domain algebra for flat relations, is discussed. In section 2.2 we present the nested relational model, which extends the flat relational model in order to allow the representation of hierarchically structured data. Research into the nested relational model began in the late 1970's [Mak77] and in recent years a lot of effort has been expended in order to further develop this model [SS87, AFS89]. In section 2.3, we describe the approach used in our implementation to support relational algebra and domain algebra for relations with arbitrary level of nesting.

2.1 The Flat Relational Model

The relational model of data was introduced by Codd [Cod70, Cod79]. It is based on a simple and uniform data structure, the relation, and has a solid theoretical foundation. The relational model is also becoming firmly established in the database application world: the majority of the commercial database management systems is based on the relational data model.

2.1.1 Basic Concepts

The relational data model represents the database as a collection of relations. Informally, each relation resembles a table or, to some extent, a simple file. Each row in the table represents a collection of related data values. In relational model terminology, a row is called a tuple, a column header is called an attribute. The data type describing the types of values that can appear in each column is called a domain. Formally, a relation is a subset of the Cartesian product of its domains. The data and structure in each relation satisfies the following four properties.

- All tuples are distinct from each other, i.e., there is no duplicate tuples in a relation.
- A relation is an unordered set of tuples. The order of tuples is unimportant.
- Each attribute is unique and the order of columns is irrelevant.
- The attribute values are atomic. By atomic we mean that each value in the domain is indivisible as far as the relational model is concerned.

The forth property, which states that the domains of attributes must include only atomic (simple, indivisible) values and that the value of any attribute in a tuple must be a single value from the domain of that attribute, is the so called first-normal-form (1NF) requirement. A relation that is in 1NF is also called a flat relation.

2.1.2 The Flat Relational Algebra

The relational algebra is a collection of operations that are used to manipulate entire relations. These operations are used, for example, to select tuples from individual relations and to combine related tuples from several relations for the purpose of specifying a query on the database. The result of each operation is a new relation, which can be further manipulated. The property that any relational algebra operation evaluates to a relation is also called the *closure principle* of the relational algebra. The *closure principle* allows complex relational expressions by concatenating a series of simple operations. The operations in the relational algebra do not change the operand relations. This property is often called the *functional principle.*

The relational algebra operations are usually divided into two groups. One group includes set operations from mathematical set theory. These are applicable because each relation is defined to be a set of tuples. Set operations include union, intersection, difference, and Cartesian product. The other group consists of operations developed specifically for relational databases. These include select, project, and join, among others.

2.1.3 Relix

Relix is an experimental relational database system developed at McGill University in 1986 [Lal86]. The data manipulation language of Relix is Aldat, an algebraic data language proposed by Merrett [Mer77]. The purpose of Relix is to provide an interactive environment of Aldat for exploring the concept of the relational database model as described in [Mer84].

Relix is implemented in C. It consists of three main modules: a lexical analyzer, a parser, and an interpreter. The lexical analyzer is generated by Lex [Les75] to scan the user input into tokens. The parser is generated by Yacc [Joh75] to perform syntax analysis. The intermediate code (I-code) is generated by invoking a translator function embedded in the parser. The interpreter interprets the I-code by calling other low level functions to perform the operations. Figure 2.1 shows the control flow of Relix.

Relix supports the relational algebra operations including selection, projection, μ-join and σ-join. The μ-join which includes natural join, union join, symmetric difference join, left join, right join, left difference join, and right difference join, are applied to relations that are union compatible. The σ-join which includes division (supset $\supseteq$), proper supset ($\supset$), equal set ($=$), proper subset ($\subset$), subset ($\subseteq$), intersection ($\cap$), and the negation of them.

Relix also supports the domain algebra operations proposed by Merrett [Mer76].

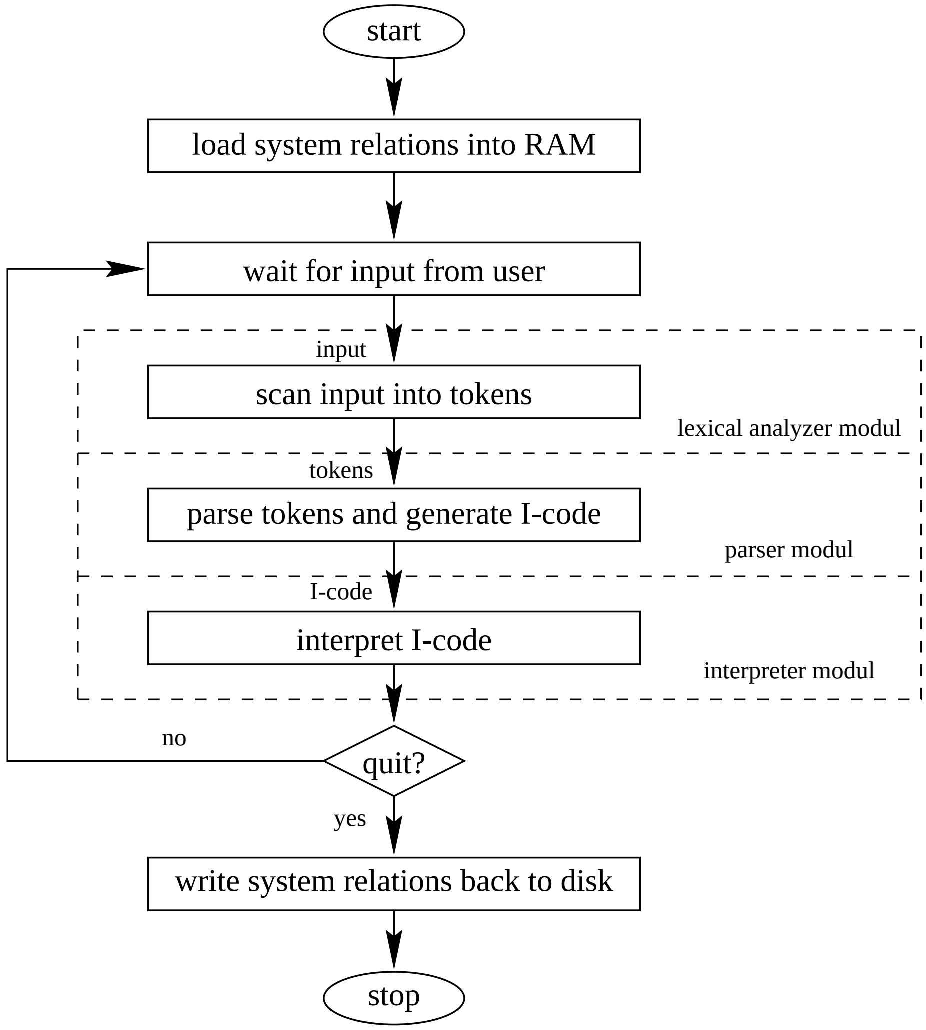


Figure 2.1: Relix execution flowchat.

Domain algebra consists of a set of operations to manipulate attributes such as mathematical operations, attribute grouping and ordering. Domain algebra is used through the declaration of virtual attributes and the actualization of them on relations. Domain algebra is divided into two categories: horizontal and vertical operations. The horizontal operations involve operations on attributes that can be actualized in a tuple-by-tuple fashion. The horizontal domain expressions are formed by applying mathematical operators, predefined functions, if-then-else clause on constants and attribute names. Vertical operations include simple reduction, equivalence reduction, functional mapping, and partial functional mapping. Simple reduction produces a single result from the values from all tuples of a single attribute in the relation. Equivalence reduction produces a different result for different sets of tuples in the relation. Each set is characterized by all tuples having the same value for some specified grouping attributes. Functional mapping processes the tuples in the order specified by some controlling attributes. Partial functional mapping performs functional mapping within each set of tuples characterized by some grouping attributes.

Besides the relational algebra operations and the domain algebra operations, Relix also provides mechanisms to update relations, to declare relations and attributes, and to create relations using relational expressions. Commands used to show the relations and attributes, to print the content of relations, to delete relations and attributes, etc, are also provided for the user to interact with the interpreter. Reference [Mer84] contains the detailed description of the relational and domain algebra in Relix. Recently, the domain algebra for relations with one-level of nesting has also been implemented [He97].

2.2 The Nested Relational Model

The nested relational model [Mak77, SS87, AFS89] was developed in order to extend the applicability of the flat relational model to non-business applications. Such applications include office automation, computer aided design, image processing, spatial data, text retrieval, expert systems, and geographical and statistical analysis.

2.2.1 Nested Relations

The nested relational model removes the 1NF restriction from the flat relational model, and allows attribute values to be relation-valued as well as atomic. Thus in the nested relational model attribute domains are defined recursively, i.e., an element of a relation-valued domain may be itself relation-valued. The resulting non-first-normal-form relations (NF^2) are called nested relations and are also referred to in the literature as complex objects [SS87, AFS89].

The main advantages that result from the use of nested relations in comparison to flat relations are the following:

- Nested relations minimize redundancy data.
- Query processing is more efficient, since some of the joins are realized with the nested relations themselves.
- Nested relations allow a very flexible interface at the external level, since both flat and hierarchical data can be presented to the user.
- Nested relations provide for the explicit representation of the semantics of the application.

Due to the extensive research in the field, significant progress has been made in the field of nested relations since the nested relational model was first proposed in 1977 [Mak77]. Fisher and Van Gucht [FG85] discussed the one-level nested relations and developed a polynomial time algorithm to test if a structure is an one-level nested relation. Thomas and Fischer[TF86] generalized the one-level nested relations to nested relations of arbitrary but finite depth. The definition of recursively nested relations was also discussed [LS88]. Various query languages have been introduced for the nested relational model, and extensions have been proposed to practical query languages such as SQL to accommodate nested relations [PA86, KR89]. Implementations [DG88, SAB+89] of database management systems based on the nested relational model are also available.

2.2.2 The NEST and UNNEST Operators

In most of the researches on nested relations, particular attention is paid to extending the flat relational model to the nested relational model and introducing two restructuring operators, NEST and UNNEST [JS82, FT83] to transform between flat and nested relations. The NEST operator creates partitions which are based on the formation of equivalence classes. Tuples are equivalent on some attributes if the values of these attributes are the same. All equivalent tuples are replaced with a single tuple in the resulting nested relation. The resulting relation is then defined on all the attributes on which the equivalence is defined, and a nested attribute that is defined on the rest of the attributes in the original relation. The UNNEST operator reverses the effect of NEST operator. Intuitively, nesting transforms a nested relation into one which is more deeply nested, while unnesting transforms a nested relation into a flatter one.

A NEST followed with an UNNEST always gives the original relation. But the reverse is not true, since applying UNNEST first may result duplicate tuples which are eliminated before the successive NEST is applied.

The NEST and UNNEST operators used to restructure the nested relations allow us to view the information in the nested database in different ways by changing the structure of the nested relations. That is, a nested relation generally has alternative representation of data which is uniquely represented by a 1NF relation.

There are two different assumptions [Tak89] with respect to whether these alternative representations are the same. The first one [Mak77] takes the structure of the representation into account, since it catches a certain semantics of the application. It follows that each nested relation should be recognized as a unique representation of data. This assumption, however, has a drawback that we lose information when it is normalized into a set of simple tuples. This assumption poses a semantic gap between 1NF and nested form relations although it enables us to represent complex objects in a natural way using nested relations. The second one, on the contrary, assumes that each set of values is just a union of single values rather than a specific object.

This assumption allows us to identify the different nested representations with their unique 1NF relation. In fact, this is an implicit assumption of many research papers such as transformation between 1NF relations and nested relations using NEST and UNNEST operators [JS82, FT83], designing nested relations [OY87], and data manipulation [AMM83].

2.3 The Approach in the Thesis

The implementation in this thesis is based on the syntax and functionalities for flat relations in Relix. Instead of concentrating on some special operations on nested relations such as nesting and unnesting, we redesign the whole system and integrate the operations for nested relations into those for flat relations through four mechanisms. The first is the introduction of new syntax for the declaration of nested attributes and relations, the initialization of nested relations, and the nested update. The second is the integration of relational expressions into domain expressions so that relational operations can be applied to nested attributes. The third is the definition of order between relations so that relation-valued attributes are treated the same way as attributes of primitive type. The forth is to use a set of flat relations which are linked to each other by surrogates, to model a nested relation. In our approach, we think that the structure of the nested relations represents part of the semantics of the application. The order between relations is defined to be dependent on the structure of relations so that an efficient algorithm for finding the order is possible. With these mechanisms, our implementation provides a uniform syntax and notation for both flat and nested relations, and supports relational algebra for nested relations with arbitrary level of nesting, and domain algebra for nested attributes at any level of the nesting hierarchy.

Chapter 3

Users' Manual on Nested Relations

In this chapter, we describe how to use relational algebra, assignment and incremental assignment, update, and views for both flat and nested relations in our Java implementation of Relix. Domain algebra is also briefly presented. We implement only a large subset of the grammar specification in appendix A. The basic syntax and semantics in our implementation follow closely those of former Relix, with additional syntax added to accommodate nested relations. There are three significant changes in the syntax. The first is the introduction of new syntax for the declaration of nested attributes and relations, and the initialization of nested relations. The second is the integration of relational expressions into domain expressions so that nested attributes are treated the same way as attributes of primitive type. The third is the nested update for nested attributes in nested relations. Computation is redesigned and will be addressed in [Bak98].

In section 3.1, we show how to start the interactive interpreter. In section 3.2, we describe the lexical conventions of the interactive language. In section 3.4, we describe the declaration of attributes and relations, and the way to initialize the relations. In section 3.3, the usage of commands are summarized. In section 3.5, we describe the relational expressions (selection, projection, and join), assignment and incremental assignment, and update. In section 3.6, we present the domain algebra for flat and nested relations. In section 3.7, we show how to use views.

Throughout this chapter, we use sample relations to show the way to declare, query and update relations.

3.1 Getting Started

The interpreter runs on any platform which has the Java Runtime Environment (JRE) version 1.1. The typical installation is to pack all the compiled class files into a file using tool `zip` or `jar` (Java archive), and add this file to the Java environment variable CLASSPATH. To start the interpreter, simply type[1]

`java` [*options*] `Interpreter`

The *options* can be any valid option invoked with the java interpreter `java`. One common option is to set the initial heap size of the Java virtual machine to the maximum amount of RAM available to the database interpreter.

After starting up, the interpreter checks the existence of some database system files. If these system files exist, the interpreter builds system catalog from these files. Otherwise new system files are created for the new database. Then the prompt > is shown and user input is expected.

The parsing unit is a statement or a command that followed by a semicolon. Statements may be put together to form a complex statement, such as the statement block used in computation.

There are three phases in the interpretation of a statement or command. They are parsing and syntactic check phase, semantic check phase, and execution phase.

[1]In this chapter, `typewriter` font is used for keywords which are typed as they are, *italic* font is used for non-terminal production rule in grammar. Font for the name of a relation or an attribute is the same as that for the ordinary text. The convention used to write Backus-Naur form syntax is defined in table A.1.

3.2 Lexical Conventions

In the parsing and syntactic check phase, the user input is reduced to a sequence of tokens. All the tokens are case-sensitive, i.e., upper and lower case letters are different.

3.2.1 Tokens

There are five classes of tokens : identifiers, keywords, operators and functions, constants and literals, and other separators. White spaces including blanks, tabs, newlines, formfeeds and comments are ignored except as they separate tokens.

Two consecutive left angle brackets $<<$ introduce a comment, which terminates to the end of line.

3.2.2 Identifiers

An identifier is an ordinary identifier or an ordinary identifier with a dot . as prefix. An ordinary identifier is a sequence of letters, digits, underscores, single quotes. The first character of an ordinary identifier must be a letter. There is no limit for the length of an identifier.

3.2.3 Keywords

The identifiers shown in table 3.1 are reserved for use as keywords, and may not be used otherwise.

3.2.4 Operators and Functions

Table 3.2 summarizes the rules for precedence and associativity of operators used to construct relational and domain expressions. Operators on the same line have the same precedence; rows are in order of increasing precedence; if not specified, they are binary operators. The ordered operators `pred` and `succ` are used only with functional

`abs`	`acos`	`add`	`also`	`alt`	`and`
`asin`	`atan`	`be`	`bool`	`boolean`	`by`
`cat`	`ceil`	`change`	`chr`	`comp`	`computation`
`cos`	`cosh`	`dc`	`deadlock`	`debug`	`deld`
`delete`	`delr`	`div`	`djoin`	`dk`	`dljoin`
`do`	`domain`	`double`	`drjoin`	`edit`	`else`
`eqjoin`	`equiv`	`eval`	`exec`	`exit`	`expr`
`expression`	`false`	`float`	`floor`	`for`	`from`
`fun`	`gejoin`	`gtjoin`	`help`	`icomp`	`iejoin`
`if`	`ijoin`	`in`	`initial`	`input`	`integer`
`intg`	`is`	`isknown`	`lejoin`	`let`	`ljoin`
`local`	`log`	`log10`	`long`	`loop`	`ltjoin`
`max`	`min`	`mod`	`natcomp`	`natjoin`	`nop`
`not`	`null`	`of`	`or`	`ord`	`order`
`out`	`par`	`pick`	`pr`	`pred`	`print`
`quant`	`quit`	`rcp`	`real`	`red`	`relation`
`rjoin`	`round`	`sd`	`sep`	`seq`	`short`
`sin`	`sinh`	`sjoin`	`sqrt`	`sr`	`srd`
`ssd`	`ssr`	`state`	`stmt`	`statement`	`strg`
`string`	`sub`	`substr`	`succ`	`sup`	`tan`
`tanh`	`target`	`text`	`then`	`time`	`to`
`true`	`ujoin`	`update`	`when`	`where`	`while`
`zorder`					

Table 3.1: List of keywords.

or partial functional mapping and are not included in the table. The unary operators #, . and % used in quantifiers are also not included in the table. Note that unary + and − have higher precedence than the binary form.

Operators	Associativity
`eval`	right to left
	unary
`or` |	left to right
`and` &	left to right
= != > < >= <= `substr`	left to right
`cat`	left to right
`max min`	left to right
+ −	left to right
join operators	left to right
projection	left to right
selection	left to right
`pick`	right to left
	unary
* / `mod`	left to right
`not` !	right to left
	unary
**	right to left
other operators	
()	left to right

Table 3.2: Precedence and associativity of operators used in expressions.

In table 3.2, *join operators* include no operation `nop`, μ-join which includes natural join (`ijoin` or `natjoin`), union join (`ujoin`), symmetric difference join (`sjoin`), left join (`ljoin`), right join (`rjoin`), left difference join (`dljoin` or `djoin`), and right difference join (`drjoin`), and σ-join which includes natural composition (`icomp` or `natcomp`), division or supset ($\supseteq$) join (`gejoin` or `div` or `sup`), proper supset ($\supset$) join (`gtjoin`), equal set ($=$) join (`eqjoin`), proper subset ($\subset$) join (`ltjoin`), subset ($\subseteq$) join (`lejoin` or `sub`), and the negation of σ-join. The *projection* and *selection* operations are discussed in section 3.5.1. *Other operators* include `if-then-else` and vertical domain operations `red`, `equiv`, `fun`, and `par`.

The assignment and incremental assignment operators $<-$ and $<+$ are used to assign an expression to a relation. The assignment operator is also used in update statement to set the new values of attributes being changed. Inside computation, it may also be used to specify the way the new attributes are computed.

Three operators in table 3.3 are used to specify the order of executing a set of statements [Kom92].

Operators	Associativity	Order of execution
$--$	left to right	sequential
$\|\|$	left to right	parallel
??	left to right	chose the unblocked one randomly

Table 3.3: Operators on statements.

The identifiers in table 3.2.4 are reserved as the predefined functions which can be applied to attributes for more sophisticated operations. The meaning of these functions is similar to their Pascal counterparts.

`abs`	`acos`	`asin`	`atan`	`ceil`
`chr`	`cos`	`cosh`	`floor`	`isknown`
`log`	`log10`	`ord`	`round`	`sin`
`sinh`	`sqrt`	`tan`	`tanh`	

Table 3.4: Functions supported in expressions.

3.2.5 Constants and Literals

There are five kinds of constants supported in our system: boolean, integer, floating, string, and relation constants. All these constants have two special null values, `dk` for don't know, and `dc` for don't care. The rules on the arithmetic of `dk` and `dc` are described in [Mer84].

Boolean Constants

Boolean constants include two valid values `true`, `false`, besides two null values `dk` and `dc`.

Integer Constants

An integer constant is a sequence of digits. It is signed decimal. It may be suffixed by the letter `s` or `S`, to specify that it is a `short` integer, by the letter `i` or `I` to specify that it is a regular `integer`, by the letter `l` or `L` to specify that it is a `long` integer. The default (without any suffix) is regular `integer`.

Floating Constants

A floating constant consists of an integer part, a decimal point, a fraction part, an optional exponent part, and an optional type suffix. The exponent part consists an `e` or `E`, and an integer exponent. The suffix is the letter `f` or `F` for `float`, or `d` or `D` for `double`. The default (without any suffix) is `float`.

String Constants

A string constant, also called string literal, is a sequence of ASCII characters surrounded by double quotes.

Relation Constants

A relation constant is a relation. The concept of relation constants is introduced because a virtual domain expression with nested attributes may contain relations at higher level (eg. top level relations). These higher level relations are constants since they are the same when the domain expression is evaluated on each tuple of the relation on which it is actualized.

3.2.6 Separators

The semicolon ; is used to terminate commands and statements. The curly bracket pair { and } is used to group a set of statements, or a set of tuples in the initialization of relations. The square bracket pair [and] is used to group attributes in projection or join. It is also used as an array shorthand for selection.

3.3 System Commands

The interactive interpreter has commands to print the content of a relation, to show and delete attributes, relations, views and computations, to turn on or turn off some system options. The following is a brief summary of the usage of commands.

`help` (*Identifier*)? - Get online help on the topic specified by *Identifier*. List all topics if option *Identifier* is absent.

`quit` - Quit the interpreter. Information on the database is saved.

`pr` *Expression* - Print the content of the result of *Expression*.

`sd` (*Identifier*)? - Show information on attribute named by *Identifier*. Show all attributes if option *Identifier* is absent.

`sr` (*Identifier*)? - Show information on relation named by *Identifier*. Show all relations if option *Identifier* is absent.

`srd` (*Identifier*)? - Show the attributes on which the relation named by *Identifier* is defined. Show the attributes of all relations if option *Identifier* is absent.

`deld` *IDList* - Delete attributes specified in *IDList*.

`delr` *IDList* - Delete relations, views or computations specified in *IDList*.

`input` *FilePath* - Redirect the input to the file specified by *FilePath*.

`time` - Toggle the interpretation timer. The clock time for interpreting each statement is displayed if the timer is on. The default is off.

`debug` - Toggle the debug mode. If the switch is on, the syntax tree of each statement and the stack trace of method calls if an exception occurs, are displayed. The default is off.

`ssd` - Toggle the hidden attributes mode. If the switch is on, the attributes start with dot . are shown using `sd` command. The default is off.

`ssr` - Toggle the hidden relations mode. If the switch is on, the relations start with dot . are shown using `sr` command. The default is off.

3.4 Domain and Relation Declaration

This section describes the syntax for the declaration of attributes and relations, and the initialization of relations at declaration time.

3.4.1 Domain Declaration

Attributes used in relations are declared with keyword `domain` using the following syntax.

`domain` *IDList Type*

Here *IDList* specifies a list of the names of the attributes being declared, and *Type* specifies the type of these attributes. Table 3.5 summarizes the valid attribute types and their domains of values.

The primitive data types `boolean, short, integer, long, float` and `double` are similiar to the corresponding Java primitive data types, except that two valid Java

Attribute Type	Alias	Values
`boolean`	`bool`	`true, false, dk, dc`
`short`		32, -57
`integer`	`intg`	87, 57330
`long`		79999444, -28
`float`	`real`	1.883, 45.88
`double`		-2.333333333
`string`	`strg`	"James Bond"
`text`		"long text"
`expression`	`expr`	"A ijoin C"
`statement`	`stmt`	"S <– [A] `in` R"
"(" *IDList* ")"		(A,B,C)
`computation` "(" *IDList* ")"	`comp`	`comp` (A,B,C)

Table 3.5: Attribute types.

values in type `short`, `integer`, `long`, `float` and `double` are reserved for representing two null values `dk` and `dc`. The reserved values for each data type are shown in appendix B. Type `string` supports charactor string up to 65533 in length, while `text` up to 4294967293 in length. Type "("*IDList*")" is used to declare nested attributes. The *IDList* in type "("*IDList*")" specifies the names of attributes on which nested attributes are defined. The current implementation requires that the attributes on which a new nested attribute is defined must be already defined. Therefore recursively defined nested attributes are not allowed.

There are three kinds of meta-type for attributes: type `expression` for relational and domain expressions; type `statement` for statements; and type `computation` for using computations as attributes. Meta-type `expression` and `statement` will be implemented in the future, while `computation` will be discussed in [Bak98].

3.4.2 Relation Declaration and Initialization

Relations are declared using the following syntax.

`relation` *IDList* "(" *IDList* ")" (*Initialization*)?

Here the first *IDList* specifies a list of names of the relations being declared, and the second *IDList* specifies the attributes on which these relations are defined. The production *Initialization* is optional. The most often used initialization is the so called curly bracket syntax in which relations start and terminate with curly brackets { and }, while tuples start and terminate with round brackets (and). The other way to initialize a relation is to use the name of another relation [2].

For a nested relation, surrogates are used to replace values of nested attributes. The actual relations for a nested attribute are stored in a separate relation with an additional attribute .id which is used to link surrogates of the attribute in its parent

[2] Other ways of initialization in former Relix are not supported because the nested relations and the binary storage format used in this implementation make it impossible to find data boundaries for a nested relation.

relation. The name of the separate relation is the name of its representing attribute, prifixed with a dot. These relations are thus called *dot relations*.

3.4.3 Two Examples

Two examples in figure 3.1 and 3.3 show how the attributes and relations are declared. The first example consists of three flat relations Student, Record, and Course. The second example consists of one nested relation Management which is defined on one flat attribute dname and two nested attributes professor and secretary. These two examples are used throughout the rest of this chapter to illustrate the syntax of our interactive language.

`domain` sno, cno, phone, mark, credit `integer`;

`domain` name, address, cname `string`;

`relation` Student(sno, name, address, phone) <− {(9643372, "John Bradley", "Bleury 3217", 8472232), (9507328, "Jim Fulton", "Clark 452", 2820834), (9633273, "Adam Smith", "University 1105", 9821054) };

`relation` Record(sno, cno, mark) <− {(9643372, 273, 78), (9643372, 302, 63), (9643372, 305, 82), (9507328, 302, 83), (9507328, 305, 87), (9633273, 305, 71), (9633273, 273, 76), (9633273, 251, 62) };

`relation` Course(cno, cname, credit) <− {(251, "Data Structure", 4), (273, "Linear Algebra", 4), (302, "Graph Theory", 3), (305, "Database System", 3), (440, "Compiler Design", 3) };

Figure 3.1: Declaration and initialization of a flat relation and its attributes.

Note that in relation Management in figure 3.4, surrogates 1, 3 and 5 for nested attribute professor link the values 1, 3 and 5 of attribute .id in relation .professor, while surrogates 2, 4 and 6 for nested attribute secretary link the values 2, 4 and 6 of attribute .id in relation .secretary.

In initialization, numeric data items of type `short, integer, long, float` or `double`, are converted automatically to the values of their attribute types, using the

Student			
sno	name	address	phone
9507328	Jim Fulton	Clark 452	2820834
9633273	Adam Smith	University 1105	9821054
9643372	John Bradley	Bleury 3217	8472232

Record		
sno	cno	mark
9507328	302	83
9507328	305	87
9633273	251	62
9633273	273	76
9633273	305	71
9643372	273	78
9643372	302	63
9643372	305	82

Course		
cno	cname	credit
251	Data Structure	4
273	Linear Algebra	4
302	Graph Theory	3
305	Database System	3
440	Compiler Design	3

Figure 3.2: Contents of three relations, Student, Record and Course.

`domain` dname, name `string`;

`domain` salary `integer`;

`domain` professor, secretary (name, salary);

`relation` Management(dname, professor, secretary) <− { ("Finance", {("Dick Davidson", 58), ("Charles Cooper", 72), ("Franklin Wong", 48)}, {("Joyce English", 36), ("Alicia Zelaya", 33)}), ("Accounting", {("John Smith", 53), ("Franklin Wong", 48)}, {("Nancy Clinton", 36)}), ("Marketing", {("Jennifer Wallace", 63), ("Franklin Wong", 48)}, {("Sharon Grant", 28), ("Paula Baker", 35)}) };

Figure 3.3: Declaration and initialization of a nested relation and its attributes.

convertion rules defined in Java.

The other mechanism of initialization is to use the name of another relation. For example, we may declare relation StudentBackup and initialize it with relation Student[3].

`relation` StudentBackup(sno, name, address, phone) <− Student;

3.5 Relational Algebra

In this section, we describe the syntax and semantics of the relational algebra. We start with relational expressions which include such building blocks as projection, selection, and join. Then we show how to use assignment and incremental assignment, and update. Implemented features are shown by example queries.

3.5.1 Relational Expressions

Relational expressions include projection, selection, join and other constructs. According to the closure principle of relational model, any relational expression evaluates to a relation. This allows the arbitrary combinations of the primary constructs to form complex expressions used in complicated queries.

[3]The simpler way to do this is to use the assignment syntax: StudentBackup <− Student;

Management		
dname	professor	secretary
Accounting	3	4
Finance	1	2
Marketing	5	6

.professor		
.id	name	salary
1	Charles Cooper	72
1	Dick Davidson	58
1	Franklin Wong	48
3	Franklin Wong	48
3	John Smith	53
5	Franklin Wong	48
5	Jennifer Wallace	63

.secretary		
.id	name	salary
2	Alicia Zelaya	33
2	Joyce English	36
4	Nancy Clinton	36
6	Paula Baker	35
6	Sharon Grant	28

Figure 3.4: Contents of relation Management, and two underlying *dot relations* .professor and .secretary.

Projection

Projection extracts a subset of the attributes in a relation. Duplicate tuples in the result relation are eliminated. The formal syntax for projection is as follow.

"[" (*ExpressionList*)? "]" (`in` *Projection* | *Selection*)

Here *ExpressionList* specifies the list of attributes being projected. We only implemented a subset of this syntax, i.e., the expressions in the production *ExpressionList* must be of type *Identifier*. The special case for empty attribute list, i.e., *ExpressionList* is absent, results a relation with one boolean attribute .bool and one tuple. The value is `true` if the resulting relation on which the project is performed has at least one tuple, and `false` otherwise.

Projection of nested attributes in a nested relation is the same as in a flat relation. Figure 3.5 and 3.6 show examples of projection on flat relations. Figure 3.7 shows projection of the attributes dname and professor in nested relation Management.

`Query 1`

Retrieve the attributes name and phone in relation Student.

[name, phone] `in` Student	
name	phone
Adam Smith	9821054
Jim Fulton	2820834
John Bradley	8472232

Figure 3.5: Query 1: Retrieve the attributes name and phone in relation Student.

Selection

Selection is used to select tuples which satisfy certain conditions in a relation. The syntax of selection is

(`where` | `when`) *Expression* (`in` | `from`) *Projection*

`Query 2`

Check if there is any tuple in relation Record.

[] `in` Record
.bool
true

Figure 3.6: Query 2: Check if there is any tuple in relation Record.

`Query 3`

Retrieve the attributes dname and professor in relation Management.

[dname, professor] `in` Management	
dname	professor
Accounting	3
Finance	1
Marketing	5

Figure 3.7: Query 3: Retrieve the attributes dname and professor in relation Management.

Tuples in which the conditional expression *Expression* is true are selected. The keyword `when` provides a synchronization primitive for the multiuser environment [Dou91]. The keyword `from` combines selection and update together: the selected tuples are deleted from the result relation of *Projection* if it is an updateable relation. These two additional features will be implemented in the future. Query 4 in figure 3.8 retrieves the records for student Jim Fulton in relation Student using selection.

`Query 4`

Retrieve the tuples for student Jim Fulton in relation Student.

`where` name = "Jim Fulton" `in` Student			
sno	name	address	phone
9507328	Jim Fulton	Clark 452	2820834

Figure 3.8: Query 4: Retrieve the tuples for student Jim Fulton in relation Student.

Projection and selection can be combined together in a single expression called T-select. The syntax for T-select is shown in Query 5.

`Query 5`

Retrieve the name and phone where the student number sno equals to 9507328 in relation Student.

[name, phone] `where` sno = 9507328 `in` Student	
name	phone
Jim Fulton	2820834

Figure 3.9: Query 5: Retrieve the name and phone where the student number sno equals to 9507328 in relation Student.

Join Two Relations Together

A database is decomposed into a set of relations according to the dependence between data, so that the update and maintenance are easy to manage. When a query requires information from more than one relations, join operations are necessary to put relations together. Various joins specify different conditions on which the two relations are joined together. The syntax for join is

$$\textit{Projection}\ (\textit{JoinOperator Projection} \mid \text{“[”}\,\textit{ExpressionList}\ \text{“:”}\ \textit{JoinOperator}\ (\text{“:”})?\ \textit{ExpressionList}\ \text{“]”}\ \textit{Projection})^*$$

Here *JoinOperator* includes `nop` for no operation , μ-join which includes `ijoin` or `natjoin` for natural join, `ujoin` for union join, `sjoin` for symmetric difference join, `ljoin` for left join, `rjoin` for right join, `dljoin` or `djoin` for left difference join, and `drjoin` for right difference join, and σ-join which includes `icomp` or `natcomp` for natural composition, `gejoin` or `div` or `sup` for division or supset ($\supseteq$) join, `gtjoin` for proper supset ($\supset$) join, `eqjoin` for equal set ($=$) join, `ltjoin` for proper subset ($\subset$) join, `lejoin` or `sub` for subset ($\subseteq$) join, and the negation of σ-join. The meaning of these join operators is explained in [Mer84]. All the join operators in μ-join and `icomp` (`natcomp`) in σ-join are implemented. In the first production, the common attributes of the left and the right side relations are used as join attributes. The case for no common attribute is handled specially: `ijoin`, `ujoin`, `ljoin`, and `rjoin` give the Catesian product of the two operand relations, while `sjoin`, `dljoin`, and `drjoin` result an empty relation. In the second production, *ExpressionList* specifies the list of join attributes. We only implement a subset of this syntax, i.e., the expressions in the *ExpressionList* must be of type *Identifier*.

Figure 3.10 and 3.11 show two queries using `ijoin`.

For join on nested attributes, the rule is the same as for join on attributes of primitive types: the attribute values which are relations are used in the comparison. The following rules are used to compare two relations.

1. the relation with less number of tuples is smaller, the relation with more number of tuples is larger

2. for two relations with the same number of tuples, sort the relations and then compare each tuple in the two relations until non-equality is found

`Query 6`

Find all the courses student Adam Smith ever took.

[name, cno] `where` name = "Adam Smith" `in` (Student `ijoin` Record)	
name	cno
Adam Smith	251
Adam Smith	273
Adam Smith	305

Figure 3.10: Query 6: Find all the courses student Adam Smith ever took.

`Query 7`

Find all students who ever took course 302.

[sno, name] `where` cno = 302 `in` (Student `ijoin` Record)	
sno	name
9507328	Jim Fulton
9643372	John Bradley

Figure 3.11: Query 7: Find all students who ever took course 302.

When a nested attribute is joined with attribute .id in its corresponding *dot relation*, the surrogate values themselves are used in comparison. This feature may be used to unnest nested relations using `icomp`. Query 8 in figure 3.12 shows an example to unnest the nested attribute professor in relation Management.

`Query 8`

Find the name and salary of professors in each department in relation Management.

[dname, name, salary] `in` (Management[professor :`icomp`: .id].professor)		
dname	name	salary
Accounting	Franklin Wong	48
Accounting	John Smith	53
Finance	Charles Cooper	72
Finance	Dick Davidson	58
Finance	Franklin Wong	48
Marketing	Franklin Wong	48
Marketing	Jennifer Wallace	63

Figure 3.12: Query 8: Find the name and salary of professors in each department in relation Management.

Other Expressions

`Pick` randomly picks one tuple from a relation and returns a relation with that one tuple. It does not change the operand relation. This feature may be used to create sample for statistical use. Query 9 in figure 3.13 shows the use of pick.

There are other legitimate relational expressions using operators listed in table 3.2 on scalars [ML89] such as R + S if R and S are scalars defined on numeric attributes. These are mostly desirable in computation and will be implemented later.

According to the closure principle of relational expressions in the relational data model, the above primary expressions may be combined to form complex expressions. Some of these combinations are already shown in Query 6, 7, 8 and 9.

`Query 9`

Find the courses and marks of one student.

[name, cno, mark] `in` (Record `ijoin` `pick` Student)		
name	cno	mark
John Bradley	273	78
John Bradley	302	63
John Bradley	305	82

Figure 3.13: Query 9: Find the courses and marks of one student.

3.5.2 Assignment and Incremental Assignment

Assignment (<−) creates a relation using the result of a relational expression, while incremental assignment (<+) adds the result of a relational expression to an existing relation. The syntax for assignment and incremental assignment is

Identifier (<− | <+) *Expression*

|

Identifier "[" *IDList* (<− | <+) *ExpressionList* "]" *Expression*

For assignment (<−), the first syntax always creates a relation named by the *Identifier*, no matter whether it exists or not, while the second syntax renames attributes. For incremental assignment (<+), the result of the right side *Expression* is added to the left side relation if they are defined on the same set of attributes in the first syntax, while renaming is performed for the second syntax. The incremental assignment is virtually the same as one of the update statements in section 3.5.3. Figure 3.14 is an example of assignment, and figure 3.15 is an example of incremental assignment.

3.5.3 Update

Update provides the mechanism for changing a relation locally. There are three basic update operations on relations: add, delete, and change. The syntax for update is

`Query 10`

Create a relation that contains attributes name and phone in relation Student.

NamePhone <− [name, phone] `in` Student;

NamePhone[name, phone <− name, phone] `in` Student;

NamePhone	
name	phone
Adam Smith	9821054
Jim Fulton	2820834
John Bradley	8472232

Figure 3.14: Query 10: Create a relation that contains attributes name and phone in relation Student.

`Query 11`

Add the new records for course 440 to relation Record.

`relation` Record440(sno, cno, mark) <− {(9507328, 440, 68), (9633273, 440, 83)};

Record <+ Record440;

Record		
sno	cno	mark
9507328	302	83
9507328	305	87
9507328	440	68
9633273	251	62
9633273	273	76
9633273	305	71
9633273	440	83
9643372	273	78
9643372	302	63
9643372	305	82

Figure 3.15: Query 11: Add the new records for course 440 to relation Record.

`update` *Identifier* (`add` | `delete`) *Expression*

|

`update` *Identifier* "[" *IDList* (`add` | `delete`) *IDList* "]" *Expression*

|

`update` *Identifier* `change` (*StatementList*)? (*UsingClause*)?

where *Identifier* specifies the name of the relation being updated, the optional production *UsingClause* is defined as

`using` *JoinOperator Expression*

|

`using` "[" *IDList* ":" *JoinOperator* ":" *ExpressionList* "]" *Expression*

Here the first two productions add or delete the result of *Expression* to or from the relation being updated. The semantics of `add` is the same as that of the incremental assignment. The third production updates part of the relation in the way specified by *StatementList*. The part of the relation be updated is the join result (specified by *JoinOperator*) of the relation being updated and the result of expression in *UsingClause*. If the *UsingClause* is absent, all the relation is updated.

Figure 3.16 is an example using `delete` in update to delete record for course 440 from relation Record.

Figure 3.17 shows an update which adds 5 to all the marks of course 273, using keyword `change` in update. The *StatementList* in this case contains an assignment statement.

The *StatementList* that follows the keyword `change` may contain update statements[4]. This nested update is used to update values of nested attributes in nested relations. Query 14 in figure 3.18 shows the way the nested update is used to increase the salary of secretary Alicia Zelaya in nested relation Management. The outer update locates the Finance department, while the inner update locates Alicia Zelaya in the nested attribute secretary and increases her salary by 2.

[4]The *StatementList* may also contain computation calls [Bak98].

`Query 12`

Delete the records for course 440 from relation Record.

`update` Record `delete` Record440;

Record		
sno	cno	mark
9507328	302	83
9507328	305	87
9633273	251	62
9633273	273	76
9633273	305	71
9643372	273	78
9643372	302	63
9643372	305	82

Figure 3.16: Query 12: Delete the records for course 440 from relation Record.

3.6 Domain Algebra

Domain algebra [Mer77, Mer84] consists of a set of operations to manipulate attributes such as mathematical operations, attribute grouping and ordering. Domain algebra is used through the declaration of virtual attributes and the actualization of them on relations. The detailed explanations of domain algebra and its implementation will be given in [Yua98]. Here we only give a briefly account on domain algebra operations to show the uniform syntax in our implementation.

The formal syntax for declaring a virtual attribute is

`let` *Identifier* `be` *Expression*

Here the *Identifier* specifies the name of the virtual attribute, and *Expression* is an arbitrary combination of operators, functions and constants defined in section 3.2, and the names of attributes in the relation on which the virtual attribute is to be

`Query 13`

Add 5 to all the marks of course 273.

`update` Record

`change` mark <− mark + 5

`using ijoin where` cno = 273 `in` Record;

Record		
sno	cno	mark
9507328	302	83
9507328	305	87
9633273	251	62
9633273	273	81
9633273	305	71
9643372	273	83
9643372	302	63
9643372	305	82

Figure 3.17: Query 13: Add 5 to all the marks of course 273.

`Query 14`

Increase 2 to the salary of secretary Alicia Zelaya in department Finance in relation Management.

`update` Management `change`

`update` secretary `change`

salary <− salary + 2

`using ijoin where` name = "Alicia Zelaya" `in` secretary

`using ijoin where` dname = "Finance" `in` Management;

Figure 3.18: Query 14: Increase 2 to the salary of secretary Alicia Zelaya in department Finance in relation Management.

actualized. Domain algebra operations are divided into horizontal operations and vertical operations.

3.6.1 Horizontal Operations

Virtual attributes with horizontal operations involve operations on attributes that can be actualized in a tuple-by-tuple fashion. The horizontal domain expressions are formed by applying operators except for `red`, `equiv`, `fun`, and `par`, predefined functions, if-then-else expression, to constants and attributes. Figure 3.19 and 3.20 show examples of horizontal operations.

3.6.2 Vertical Operations

Vertical operations include reduction, equivalence reduction, functional mapping, and partial functional mapping. These four vertical operations are primary expressions that can be embeded in other expressions. Reference [Mer84] defines the semantics of these operations. The syntax of them is shown in appendix A. The current implementation includes only reduction and equivalence reduction.

Simple reduction produces a single result from the values from all tuples of a single attribute in the relation. Figure 3.21 shows the query to find all professors in relation Management by union joining all the nested attribute professor in each tuple using reduction.

Equivalence reduction provides a grouping mechanism not present in simple reduction. It produces a different result for different sets of tuples in the relation. Each set is characterized by all tuples having the same value for some specified attribute. Figure 3.22 shows a query using equivalence reduction to find the average mark of each course in relation Record.

Horizontal and vertical operations may be combined together to produce quite sophisticated queries that require complicated operations over the data. Figure 3.23 shows how to calculate the grade point average (GPA) of each student, using both horizontal operations and equivalence reduction.

`Query 15`

Create a nested relation that contains the staff of each department in relation Management.

`let` staff `be` professor `ujoin` secretary;

ManagementStaff <− [dname, staff] `in` Management;

ManagementStaff	
dname	staff
Accounting	7
Finance	8
Marketing	9

.staff		
.id	name	salary
7	Franklin Wong	48
7	John Smith	53
7	Nancy Clinton	36
8	Alicia Zelaya	33
8	Charles Cooper	72
8	Dick Davidson	58
8	Franklin Wong	48
8	Joyce English	36
9	Franklin Wong	48
9	Jennifer Wallace	63
9	Paula Baker	35
9	Sharon Grant	28

Figure 3.19: Query 15: Create a nested relation that contains the staff of each department in relation Management.

`Query 16`

Create a nested relation that contains the professors with salary higher than 60 in each department in relation Management.

`let` richprof `be` [name, salary] `where` salary > 60 `in` professor;

RichProf <− [dname, richprof] `in` Management;

RichProf	
dname	richprof
Accounting	13
Finance	14
Marketing	15

.richprof		
.id	name	salary
14	Charles Cooper	72
15	Jennifer Wallace	63

Figure 3.20: Query 16: Create a nested relation that contains the professors with salary higher than 60 in each department in relation Management.

`Query 17`

Find all professors in relation Management.

`let` allprof `be red ujoin of` professor;

AllProf <− [allprof] `in` Management;

AllProf
allprof
16

.allprof		
.id	name	salary
16	Charles Cooper	72
16	Dick Davidson	58
16	Franklin Wong	48
16	Jennifer Wallace	63
16	John Smith	53

Figure 3.21: Query 17: Find all professors in relation Management.

`Query 18`

Find the average mark of each course in relation Record.

`let` avg `be` (`equiv` + `of` mark `by` cno)/(`equiv` + `of` 1 `by` cno);

RecordAVG <− [cno, avg] `in` Record;

RecordAVG	
cno	avg
251	62
273	77
302	73
305	80

Figure 3.22: Query 18: Find the average mark of each course in relation Record.

`Query 19`

Find the GPA of each student.

`let` GPA `be` (`equiv` + `of` mark*credit `by` sno)/(`equiv` + `of` credit `by` sno);

StudentGPA <− [sno, GPA] `in` (Record `ijoin` Course);

StudentGPA	
sno	GPA
9507328	85
9633273	69
9643372	74

Figure 3.23: Query 19: Find the GPA of each student.

3.6.3 Deep Level Domain Algebra

Domain algebra may be applied to relation-valued attributes in nested relations. Our system provides uniform syntax for this deep level domain algebra. Figure 3.24 shows an example to apply domain algebra to the nested attribute professor in relation Management.

3.7 View

Views are computed relations defined on relations, views and computations. A view is recursive if its expression body refers to itself directly or indirectly. At the declaration time, no actual relation is created. When the view is referred to in a relational expression, the evaluation of the view begins, using the current definition of relations, views and computations on which it is defined. The syntax for view definition is

Identifier (`initial` *Expression*)? `is` *Expression* (`target` *Expression*)?

Here *Identifier* is the name of view being declared, the `initial` and `target` clauses are optional for recursive views. The evaluation of a single recursive view starts from the result of the initial *Expression* and iterates over evaluating the view body

`Query 20`

Include sum of salaries of professors in each department in relation Management.

`let` sum `be` `red` + `of` salary;

`let` professor' `be` [name, salary, sum] `in` professor;

Management' <− [dname, professor'] `in` Management;

Management'	
dname	professor'
Accounting	25
Finance	26
Marketing	27

.professor'			
.id	name	salary	sum
25	Franklin Wong	48	101
25	John Smith	53	101
26	Charles Cooper	72	178
26	Dick Davidson	58	178
26	Franklin Wong	48	178
27	Franklin Wong	48	111
27	Jennifer Wallace	63	111

Figure 3.24: Query 20: Include sum of salaries of professors in each department in relation Management.

expression (the second *Expression*) until the result of the target *Expression* is reached or the result stops changing if there is no target clause. For mutual recursive view, see reference [CLM89] for details.

For example, if the grade point average (GPA) of students are frequently accessed, we may define a view that calculates the GPA according to the marks and credits of the courses the students ever took.

StudentGPA `is` [sno, name, GPA] `in` (Student `ijoin` Record `ijoin` Course)

Here the virtual attribute GPA is defined in Query 19.

Once it is defined, view StudentGPA can be used as if it is an ordinary relation. For example, we can find students whose GPA is higher than or equal to 70 using selection on view StudentGPA.

`Query 21`

Find students whose GPA is higher than or equal to 70.

`where` GPA >= 70 `in` StudentGPA		
sno	name	GPA
9507328	Jim Fulton	85
9643372	John Bradley	74

Figure 3.25: Query 21: Find students whose GPA is higher than or equal to 70.

Figure 3.26 and 3.27 shows the classical example for finding the transitive closure of a directed graph, using a recursive view. Figure 3.28 shows the query if a directed graph has any cycle in it.

```
domain X,Y integer;
relation G(X,Y) <- {(1,2),(1,4),(2,3),(3,5),(5,2)};
relation T(X,Y);
T is G ujoin (T[Y :icomp: X]G);
```

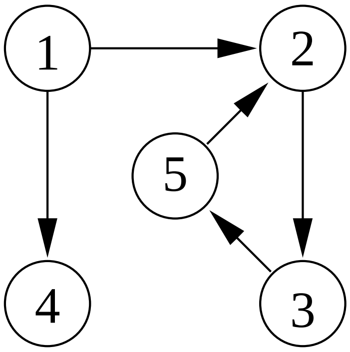


Figure 3.26: The recursive view that finds the transitive closure of a directed graph.

`Query 22`

Find all the connected pairs of nodes in directed graph G.

`pr` T;	
X	Y
1	2
1	3
1	4
1	5
2	2
2	3
2	5
3	2
3	3
3	5
5	2
5	3
5	5

Figure 3.27: Query 22: Find all the connected pairs of nodes in directed graph G.

`Query 23`

Find if the directed graph G has any cycle.

[ ] `where` X = Y `in` T
.bool
true

Figure 3.28: Query 23: Find if the directed graph G has any cycle.

Chapter 4

Implementation of Nested Relations

In this chapter we describe the Java implementation of the nested relational algebra in Relix, with particular emphasis on the way the nested relations are integrated into the frame of flat ones. In section 4.1, we give an overview of the system architecture. In section 4.2, we introduce a way to define order among relations. Once an order is defined, the relation-valued attributes can be treated the same way as the simple attributes in many of the operations on a relation, such as sort and project. The sort algorithm for nested relations, and the storage format of nested relations are presented. In section 4.3, we discuss the implementation of relational expressions, including projection, selection, μ-join, and combinations of them. In section 4.4, we describe the implementation of relational statements, including declaration and initialization of relations, assignment, update, and view. In section 4.5, we present some performance results of our implementation.

4.1 Overview

The purpose of this implementation is to integrate the notion of nested relations into both the relational algebra and the domain algebra of Relix, the current implementation of Aldat. Our implementation combines domain expressions and relational expressions together, making the domain algebra a superset of the relational algebra. This integration gives a concise grammar while supporting nested relations to any finite level[1]. The difference between relational algebra and domain algebra is that the former involves the top level relations, while the latter must be embedded in a relation which may have relations as its attribute values[2].

The new object-oriented programming language Java [AG96, GJS96] is used to implement the system. The advantages of using Java are due to its platform independence and the support of object-oriented features. The program can run on any platform which has the Java runtime environment (JRE). The support for object-oriented features and a rich set of application program interface (API) packages make programming easier, especially for the front end of our interactive system (i.e. the parser and the interpreter). For example, the store and retrieval of the code for virtual attributes, views, and computations are greatly simplified using the object serialization mechanism in Java. The big disadvantage is Java's low performance, compared with C. In fact, the object-oriented language Java is not good for implementing a database execution engine, which involves only a large number of repeated operations on simple data items.

4.1.1 System Architecture

The system contains three main parts, the parser, the interpreter, and the execution engine. The overall structure is shown in Figure 4.1. The parser parses the user input (through command lines or files) and converts it into a syntax tree if successful. The syntax tree is then passed to the interpreter. The interpreter decomposes the syntax tree into a set of low level actions and calls the methods in the execution engine for execution.

[1]The recursive nested relations are not supported.

[2]If we think that the whole database is a tuple which includes all the relations in the database as attributes, then the relational algebra may be viewed as also embedded in this database relation.

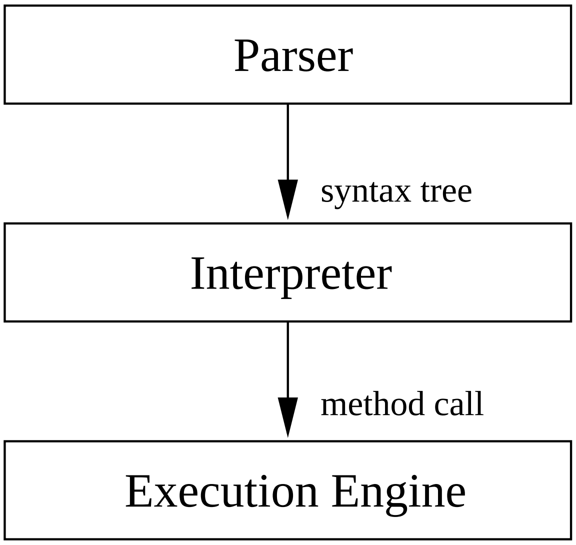


Figure 4.1: System architecture.

4.1.2 Parser

The parser is created using JavaCC [SDV], a Java compiler compiler written in Java that automatically generates parser by compiling a high-level grammar specification stored in a text file. JavaCC combines the lexical analyzer and syntactic parser together. Because JavaCC uses the top-down parsing technique [ASU86], left-recursion is not allowed in the grammar specification. Therefore the format of the grammar specification in our implementation is different from that of former Relix which uses the bottom-up parser generator Yacc [Joh75]. The Backus-Naur form of our grammar is summarized in appendix A. Note that in the grammar specification the non-terminal *Expression* is used in both the relational algebra and the domain algebra to support relational operations on the nested attributes.

The JJTree [SDV] preprocessor is used to build syntax tree while parsing. The tree building actions are carefully inserted in the JavaCC source so that syntax tree is compact and easy to interpret. Each tree node contains fields indicating the nature of the node.

Figure 4.2 and 4.3 show two sample syntax trees generated by the parser. There are two advantages of the syntax tree approach. The first is that the context is preserved,

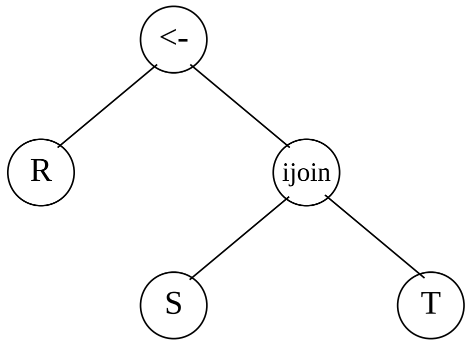


Figure 4.2: Syntax tree for assignment: R <− S `ijoin` T.

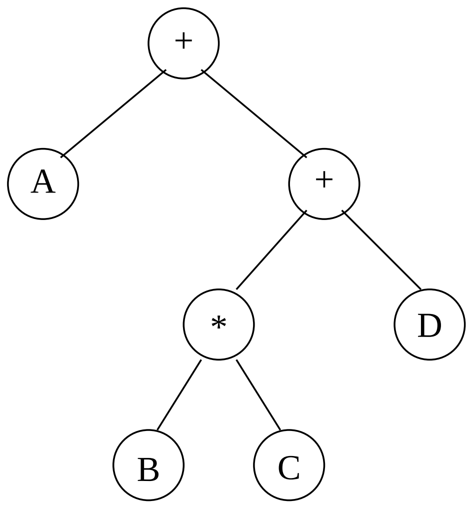


Figure 4.3: Syntax tree for expression: A + B*C + D.

making it easy to optimize the interpretation. The second is that it is efficient for interpretation compared with the I-code approach in former Relix implementation, since there is no need to maintain a separate code stack. The stack of the recursive interpreter serves as the code stack. The performance gain is very prominent in domain algebra, since the interpretation is repeated on each tuple of a relation. Also note that a pre-order traversal of the syntax tree will create stack machine code, if it is needed anyway.

4.1.3 Interpreter

The interpreter (implemented in class Interpreter.java) repeatedly calls the parser, gets the syntax tree generated by the parser, traverses the syntax tree and decomposes it into a set of method calls executed by the execution engine. The algorithms used to analyze the syntax trees are explained in section 4.3 and 4.4. The interpreter also interacts with the system tables to retrieve and update information about attributes, relations, views, and computations in the database.

4.1.4 Execution Engine

The execution engine consists of three Java classes, Relation.java, Actualizer.java and Computation.java. Class Relation encapsulates data fields representing a relation such as its name, type[3], number of tuples, list of attributes and sort status, and the methods for relational algebra such as sort, project, select and join. The implementation of these methods is explained in detail in section 4.3. The Actualizer [Yua98] actualizes virtual attributes in a relation. The Computation [Bak98] implements computation for each tuple of a relation. Figure 4.4 shows the structure of the execution engine.

There are other classes which are used to facilitate the implementation of class

[3]In our implementation, views and computations are objects of class Relation with one additional field to hold the syntax tree of code. A type field is used to differentiate them. Class Computation is a subclass of Relation so that computations can be joined with relations.

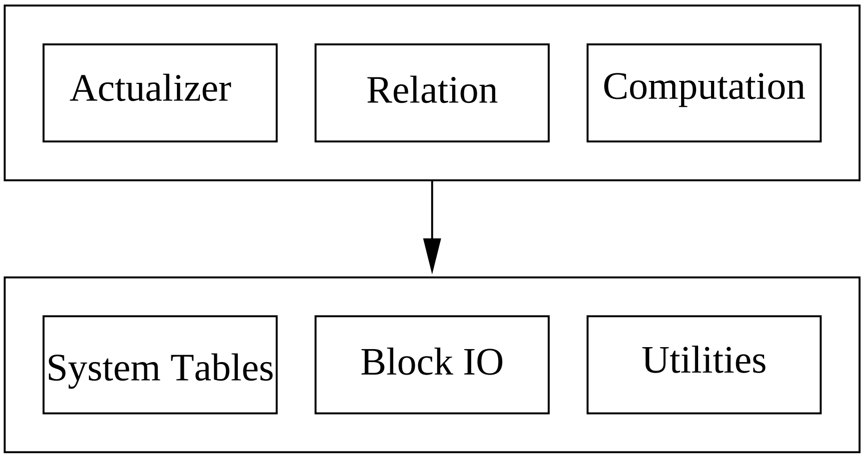


Figure 4.4: Structure of the execution engine.

Interpreter, Relation, Actualizer, and Computation. Classes BlockInputStream and BlockOutputStream provide block access to disk files. Class Surrogate creates surrogates used by nested relations. Class PrettyPrint implements methods to give formatted printout of relations.

4.1.5 System Tables and System Relations

Two system tables which are implemented in two classes DomTable.java and RelTable.java, are used to store information about attributes, relations, views, and computations in a database. Once the interpreter starts, these two system tables are constructed from three system relations (.rel, .dom, and .rd), and two files, .expr and .comp on disk. The system relations .rel and .dom store information about all the relations (including views and computations) and attributes in the database respectively, while system relation .rd stores information that links the relations with the attributes on which they are defined. Table 4.1 summarizes the attributes on which these three system relations are defined on. File .expr contains the serialized syntax trees of views and virtual attributes, while .comp contains the serialized syntax trees of computations.

Relation	Attributes	Description
.rel	.rel_name	relation name
	.tuples	number of tuples
	.attributes	number of attributes
	.rvc	type (relation, view or computation)
	.sort	number of sorted attributes
.dom	.dom_name	attribute name
	.type	attribute type (see table 3.5)
	.count	number of times this attribute is referenced
.rd	.rel_name	relation name
	.dom_name	attribute name
	.position	position of this attribute

Table 4.1: Definition of system relations .rel, .dom and .rd.

4.2 Nested Relations and Sort Algorithm

Nested relations have relation-valued attributes, in addition to simple-valued ones. We integrate the nested attributes into the frame of simple ones by defining operations among them. The operations include all those in the relational algebra. The definition of these operations makes the relational algebra applicable to nested attributes as if they are simple ones.

To make the relational operations on flat relations work also for nested relations, we have to define the meaning of `equal`, `larger-than`, and `less-than` between two relations. For example, to implement project and join efficiently, we have to sort the relations first. The sort of nested relations involves the comparisons of relation-valued attributes. In the next section, we introduce the order among relations by defining three comparative operators `equal`, `larger-than`, and `less-than`. For relation-valued attributes, the definition of `larger-than` and `less-than` is for the internal algorithm, rather than for the user[4].

4.2.1 Order of Relations

Two relations are comparable if they are defined on the same set of attributes. For two relations defined on the same set of attributes, equality is well defined, i.e., they are equal if they have the same set of tuples. Except for equality, the order of relations, i.e., the meaning of `larger-than` and `less-than`, can be arbitrarily defined. To make sure that the algorithm for projection (sort then eliminate duplicates) and the sort-merge algorithm of join are correct for nested relations, the definition of `larger-than` and `less-than` should satisfies the four properties in specified in figure 4.5.

A `less-than` $B \Longrightarrow A$ not `equal` to B

A `larger-than` $B \Longrightarrow A$ not `equal` to B

A `less-than` B and B `less-than` $C \Longrightarrow A$ `less-than` C

A `larger-than` B and B `larger-than` $C \Longrightarrow A$ `larger-than` C

Figure 4.5: Properties of comparative operators that make sort-merge algorithm work for nested relations.

We define the meaning of `larger-than` and `less-than` by the following rules.

- the relation with fewer tuples is smaller; the relation with more tuples is larger
- for two relations with same number of tuples[5], sort the relations and then compare two tuples in each relation until non-equality is found

[4]However, the user may take advantage of this definition and use it in specifying queries.

[5]In this case, the order also depends on the order of attributes on which the two relations are defined. There are two possible approaches with respect to rearranging the order of attributes. The first approach rearranges the attributes in alphabetic order, while the second one rearranges the attribute order of one relation to that of the other one. The advantage of the first approach is that it defines a canonical form of order. The second approach is more efficient, because the swapping operation is not as frequent as in the first approach. Both approaches work for the relational algebra operations. We adopt the second approach.

The above rules apparently satisfy the order properties in figure 4.5. These rules use the number of tuples of a relation as its signature. The comparison stops if the signatures are different. This can speed up the comparison since the second rule may not be invoked. Note that the above rules are recursive in nature, because the sort of two relations and the comparison of tuples in the second rule may involve comparison of nested relations one level down.

In summary, we use the recursive algorithm in figure 4.6 to compare two relations. The **Sort** and **CompareTuple** in the algorithm may recursively call **CompareRelation** itself.

CompareRelation(relA, relB)

    if relA and relB not defined on the same set of attributes

        return `non-comparable`

    else

        if (relA.tuples $<$ relB.tuples)

            return `less-than`

        else if (relA.tuples $>$ relB.tuples)

            return `larger-than`

        else

            Swap attribute order of relB to that of relA

            Sort relA and relB

            for i from 1 to relA.tuples

                CompareTuple(relA.tuple[i], relB.tuple[i])

        return `equal`

Figure 4.6: Algorithm for **CompareRelation** method. Method **Swap** swaps the columns of a relation. Method **Sort** sorts a relation.

4.2.2 Sort Algorithm

The heap sort algorithm [CLR90] is used to sort a relation on specified attributes. Heap sort is in-place sort with worst-case complexity $N \log N$, where N is the number of comparisons of attribute values. For nested relations with relation-valued attributes, there may have N comparisons of relations nested one level down. The sort method uses the method **CompareTuple** to compare two tuples, which may recursively call **CompareRelation**.

CompareTuple(tupleA, tupleB)
    for i from 1 to tupleA.attributes
        attrA ← tupleA.attribute[i]
        attrB ← tupleB.attribute[i]
        if attrA and attrB are of primitive data type
            Compare(attrA, attrB)
        else /* relation type */
            relA ← **GetRelation**(attrA)
            relB ← **GetRelation**(attrB)
            CompareRelation(relA, relB)

Figure 4.7: Algorithm for **CompareTuple**. Method **Compare** compares two values of primitive type. Method **GetRelation** returns the relation represented by a surrogate.

Our approach thus follows the hierarchical structure of a relation in comparison. The comparison always goes only one level down and compares the signatures of the operand relations. Only when the signatures are the same, does the comparison proceed one level down further. Because we take into account the structure of nested relations, we virtually follow the assumption in [Mak77] with respect to the interpretation of alternative representations of nested relations. This approach is different from the one adopted in many other research papers [JS82, FT83, AMM83, OY87].

One advantage of our approach is that it can be much faster than the other approach which unnests the operand relations down to primitive attribute type first and then compares the flattened operand relations.

4.2.3 Format of Nested Relations

Flat relations are stored on disk in tuple-by-tuple and then attribute-by-attribute fashion described in appendix B. A nested relation is represented by several flat relations. Surrogates[6] are used to replace values of nested attributes in a nested relation. The actual relation values of a nested attribute are stored in a separate relation with an additional attribute .id. The attribute .id is used to link the surrogates of the attribute in its parent relation. The names of the separate relations are the names of their attributes, prefixed with a dot. These relations are thus called *dot relations*.

For the nested relation R in figure 4.8, the relation .X is created after the declaration of the nested domain X. The surrogates 1 and 2 for nested attribute X in relation R link with the values 1 and 2 of attribute .id in relation .X.

In our implementation, *dot relations* are always sorted on the .id attribute. A binary search can be used to find the relation referred to by a surrogate.

4.3 Relational Expressions

Relational expressions are used to specify queries to the database. We adopt the following two policies in implementing the relational algebra.

- *Functional principle* – The evaluation of a relational expression does not change the operand relations in the expression.

[6]We use Java long integers to represent surrogates. Since the maximum long integer in Java is 9,223,372,036,854,775,807, it will wrap around after 292,471 years, assuming that the surrogates are generated at a speed of 1,000,000 per second. The value of next surrogate is stored on disk to make it permanent.

```
domain A, B, C integer;
domain X(B, C);
relation R(A, X) <- {(3,{(2,2),(3,3),(4,4)}), (6,{(4,4),(5,5)})};
```

R	
A	X
3	1
6	2

.X		
.id	B	C
1	2	2
1	3	3
1	4	4
2	4	4
2	5	5

Figure 4.8: Format of the nested relation R. The surrogates 1 and 2 for X in R link with the corresponding values 1 and 2 of attribute .id in relation .X.

- *Surrogate and data reuse policy* – Surrogates and their represented relations are reused wherever possible. As a result, a surrogate and its represented relation may be referred to by more than one relation. This policy eliminates the need to duplicate data in projection, selection, join and assignment. The relations represented by some surrogates may only be duplicated in the update statement. The reuse of surrogate and data can also help speed up the comparison of nested attributes, because two relations represented by the same surrogate are equal – there is no need to compare the contents of these two relations.

According to the *closure principle* of the relational model, any relational expression evaluates to a relation. The *functional principle*, together with the *closure principle*, allows arbitrary combinations of projection, selection, join, and other legal relational operations to form complex expressions for complicated queries, leaving all the operand relations unchanged.

4.3.1 Project

Projection projects out the desired attributes of a relation. Sorting on the attributes is first performed to facilitate the elimination of possible duplicates in the result due to the loss of attributes. For nested relations, simply use the old surrogates for the nested attributes in the result relation. There is no subtlety introduced at all.

4.3.2 Select

Selection is quite straightforward. A virtual attribute representing the conditional expression of the selection is constructed on the fly and is actualized on the source relation. The tuples in which the actualized conditional attribute is true are selected.

4.3.3 μ-join

μ-joins are implemented using sort-merge algorithm [Mer84]. If the join attributes are the same, the reuse of old surrogates for nested attributes imposes no problem

– the *dot relation* for a nested attribute contains the relations for all the surrogates used to represent that nested attribute.

However, if the join attributes are different, for joins other than `ijoin`, reuse of surrogates for other attributes may give rise to dangling reference – there is no underlying relations in the *dot relation* for those reused surrogates. This situation is shown in figure 4.9. In the result relation T, the surrogate 4 for nested attribute X does not have any corresponding relation in .X. The same is true for surrogate 2 for nested attribute Y.

The solution is to copy the data referred to by a surrogate to the *dot relation* of the nested attribute for which the surrogate is reused. For the example in figure 4.9, we need to copy data for surrogate 4 in relation .Y to relation .X and data for surrogate 2 in relation .X to relation .Y.

To join relations on nested attributes, relation values are used in the comparison. The surrogates are used simply as links to find the underlying relations. Sometimes we want to use the surrogate values themselves in the comparison for the join. For example, we may join a nested attribute with the .id attribute in its *dot relation* to unnest the nested relation (see Query 8 in figure 3.12). The implementation of the domain algebra for nested attributes using the top-level approach (contrary to the tuple-by-tuple approach) has to uses this feature. Therefore we introduce an additional rule for join on nested attributes – if one of them is .id attribute, the join is on the surrogate values themselves.

4.3.4 Other Expressions

Pick randomly picks one tuple from a relation and returns a relation with that one tuple. It does not change the operand relation.

There are other legitimate relational expressions for scalars [ML89] such as R + S if R and S are scalars defined on numeric attributes. These are mostly desirable in computation and will be implemented in the future.

`domain` A, B, C, D `integer`;

`domain` X, Y (A, B);

`relation` R(C, X) <− {(3,{(3,3),(4,4)}),(6,{(4,4),(5,5),(6,6)}),(6,{(3,3),(4,4)})};

`relation` S(Y, D) <− {({(2,2),(3,3)}, 5),({(3,3),(4,4)}, 6),({(3,3),(4,4)}, 12)};

T <− R [X : `sjoin` : Y] S;

R	
C	X
3	1
6	3
6	2

S	
Y	D
4	5
5	6
6	12

.X		
.id	A	B
1	3	3
1	4	4
2	4	4
2	5	5
2	6	6
3	3	3
3	4	4

.Y		
.id	A	B
4	2	2
4	3	3
5	3	3
5	4	4
6	3	3
6	4	4

T			
X	C	Y	D
4	dc	4	5
2	6	2	dc

Figure 4.9: μ-join introduces dangling reference. In the result relation T, the surrogate 4 for the nested attribute X has no corresponding relation in .X, and surrogate 2 for the nested attribute Y has no corresponding relation in .Y too.

4.3.5 Complex Expressions

All the above relational expressions can be combined to form complex expressions. The recursive method **EvaluateExpression** in figure 4.10 evaluates any complex expressions. It recursively traverses the syntax tree until operands are resolved at the leaves, which are identifiers representing relations, views, or computations. We use a hash table to pass evaluation environment. This is desirable in evaluating recursive views to pass the results from the last iteration.

4.4 Relational Statements

Statements include the declaration of relations and views, assignment and incremental assignment, update, and combinations of these simple statements, such as sequential and parallel statement, block of statement.

4.4.1 Declaration and Initialization of Relations

When a relation is declared, the interpreter checks the existence of its attributes, then instantiates an object of class Relation and adds it to the system table. The reference count of the attributes on which it is defined is increased by one so that these attributes can not be deleted or changed. The declared relation replaces the one (including relation, view, or computation) with the same name in the system table.

There are two ways to initialize a relation at the declaration time. The first one uses the name of an existing relation. In this case, the right side relation and the relation being initialized must be defined on the same set of attributes. Simply copy the attribute values from the right side relation to the corresponding attributes of the relation being initialized. For nested relations, the underlying *dot relations* are not touched, since surrogates are reused by the same nested attribute.

The second one uses the so-called curly bracket syntax. In this case, the interpreter parses the initialization tree (see figure 4.11) generated by the parser, extracts the data from the tree leaves, and puts the data into the corresponding data arrays in

```
EvaluateExpression(node, hashtable)
    switch (node.type)
        case IDENTIFIER
            if node.id is in hashtable
                return hashtable.get(node.id)
            else if node.id is a view
                view ← SystemTable.get(node.id)
                return EvaluateView(view.tree, hashtable)
            else
                return SystemTable.get(node.id)
        case JOINOPERATOR
            relA ← EvaluateExpression(node.left, hashtable)
            relB ← EvaluateExpression(node.right, hashtable)
            return Join(relA, relB)
        case PROJECT
            attributeList ← ConvertNodeToList(node.left)
            rel ← EvaluateExpression(node.right, hashtable)
            return Project(attributeList, rel)
        case SELECT
            condition ← node.left
            rel ← EvaluateExpression(node.right, hashtable)
            return Select(condition, rel)
        ...
```

Figure 4.10: Algorithm for method **EvaluateExpression** which is used to evaluate general relational expressions. Method **EvaluateView** evaluates a view, **Join** joins two relations using the sort-merge algorithm, **ConvertNodeToList** converts part of a syntax tree to a list of attributes, **Project** project a list of attributes from a relation, and **Select** selects the tuples that satisfy a certain condition from a relation. **Join**, **Project**, and **Select** are methods defined in class Relation.

RAM for the relation be initialized and the *dot relations* of its nested attributes. Before appending the data arrays to the relation and the underlying *dot relations*, duplicates are eliminated using projection.

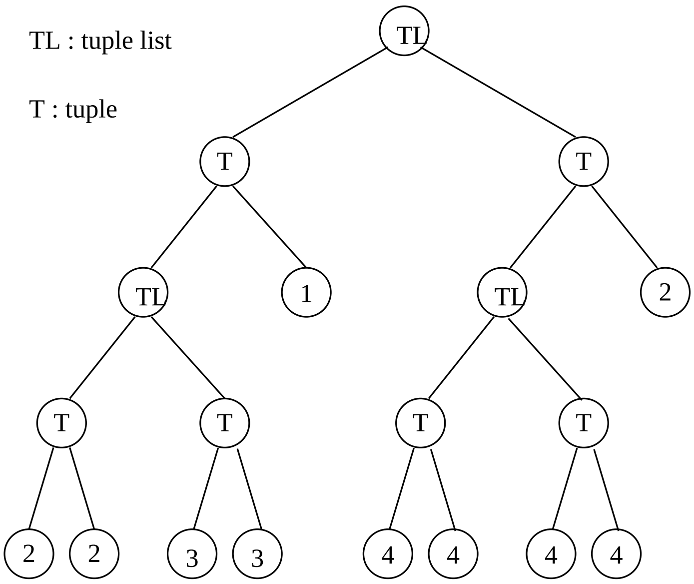


Figure 4.11: The initialization tree of curly bracket expression {({(2,2),(3,3)}, 1),({(4,4),(4,4)},2)}. Here TL indicates that the subtree is a list of tuples, while T indicates that the subtree is a tuple, namely a list of attribute values.

Parsing and extracting data from the initialization tree is done recursively. The idea is that in traversing down the tree, when a tree node of type T (tuple) is found, parse all its children to form a tuple and add the tuple to the data array for the corresponding relation or *dot relations*, when a tree node of type TL (tuple list) is found, parse each of its children which are of type T (tuple). The parsing stops going down as far as leaves are reached. All the primitive data items are stored in the leaves of the tree. The next surrogate value is used for each node of type TL (tuple list).

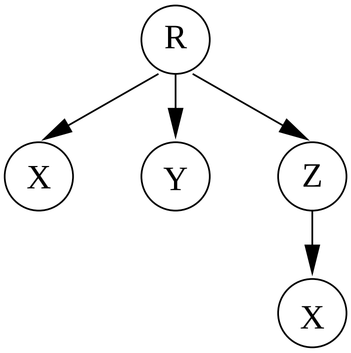
The removal of duplicates in the initialization is performed in a bottom-up fashion after parsing the initialization tree. From the dependence graph of the nested attributes, a topological sort [CLR90] gives the reverse order of the process for removing the duplicates. Figure 4.12 is an example of the dependence graph and its topological order for nested relation R. In this example, the process for removing duplicates in the temporary data arrays starts from nested attribute X, then Z, then Y, and finally relation R;

`domain` A, B, C, D `integer`;

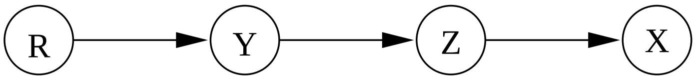
`domain` X, Y (A, B);

`domain` Z (X, C);

`relation` R (X, Y, Z, C);



(a) dependence graph of the relation schema of R



(b) a topological order of the graph in (a)

Figure 4.12: The dependence graph and its topological order of the relation schema of R: (a) dependence graph of the relation schema of R; (b) a topological order of the graph in (a).

The rule for the declaration of nested attributes, i.e., the attributes on which a nested attribute are defined must be already declared, makes sure that the dependence graph is always a directed acyclic graph (DAG). Therefore, the topological sort algorithm always works and a bottom-up path always exists.

In the bottom-up process, we know that when the higher level relations are processed, the lower level relations are sorted (because of the sort in projection for removing duplicates) and are free of duplicates. This property helps to speed up the comparisons during the process of removing duplicates in higher level relations.

All the above processing is done for the temporary data arrays in RAM. The data already in the *dot relations* of the nested attributes of the relation are not touched, since the use of new surrogates makes it impossible to have duplicates between the temporary data arrays in RAM and the ones on disk. After removing the duplicates, the temporary data array for the relation is save on disk, and the temporary data arrays for the nested attributes of the relation are appended to their corresponding files on disk.

All the initialized relations are sorted for all the attributes in the order given by the declaration as a by-product of the duplicate-removal process.

4.4.2 Assignment and Incremental Assignment

There are two kinds of syntax for assignment ($<-$) and incremental assignment ($<+$).

$$\textit{Identifier}\ (<- \mid <+)\ \textit{Expression}$$
$$\mid$$
$$\textit{Identifier}\ \text{“[”}\ \textit{IDList}\ (<- \mid <+)\ \textit{ExpressionList}\ \text{“]”}\ \textit{Expression}$$

For assignment ($<-$) in the first sytax, we create a duplicate copy of the relation resulted from evaluating the right side relational expression *Expression* and then rename it to *Identifier*. For assignment ($<-$) in the second syntax, we have to check if the attributes in the right side match those of the left side. A duplicate relation is created and the attributes of it are renamed to name specified by *IDList* in the left side. If there is renaming of nested attributes in the assignment, the relations represented by surrogates for the nested attributes in the right side must be copied to the corresponding *dot relations* of the nested attributes in the left side. The copying of relations represented by surrogates are necessary to eliminate dangling reference. This is similar to the case for μ-join in figure 4.9.

```
domain A, B, C, D integer;
domain X (A, B);
domain Y (B, A);
relation R(C, X) <- {(3,{(2,3),(4,5)}),(6,{(7,8),(8,9)})};
S[Y, D <- X, C]R;
```

Figure 4.13: Renaming of nested attributes in assignment. The result relation S has nested attribute Y. The data in relation R is copied to S, and data in .X are also copied to .Y to eliminate dangling reference.

The implementation of the incremental assignment (<+) is the same as that of the assignment, except that the old data in the left side relation is retained in incremental assignment.

4.4.3 Update

Update is the only mechanism for changing a relation locally. There are three basic update operations on relations: add, delete, and change. The syntax for update is (see appendix A)

`update` *Identifier* (`add` | `delete`) *Expression*

|

`update` *Identifier* "[" *IDList* (`add` | `delete`) *IDList* "]" *Expression*

|

`update` *Identifier* `change` (*StatementList*)? (*UsingClause*)?

The first two involve only the addition and deletion of a whole relation, thus are very easy to implement. In the third production, the *StatementList* after keyword `change` is a list of assignment statements, update statements, and computation calls. Figure 4.14 shows the syntax tree for update with `change` part.

For unnested update, i.e., there are only assignment statements in *S1* to *SN* in figure 4.14, we first actualize the right side of each assignment on the part of relation to be updated, then go through the relation and make all the changes.

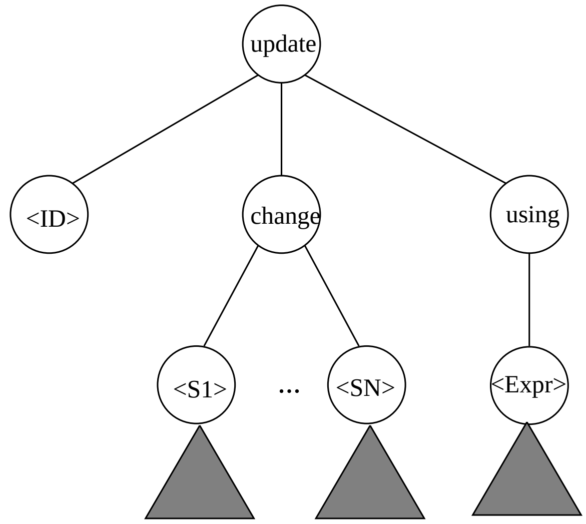


Figure 4.14: Syntax tree for update with `change` part. The statements *S1* to *SN* may be assignment statements, update statements, or computation calls.

The statements after keyword `change` may contain update statements to update nested attributes in deeper level. Figure 4.15 shows an example of nested update. We build the nested update on top of that for unnested one. The idea is to define a method that returns the updated relation and chnage the nested update statements into assignment statements. The algorithm is shown in figure 4.16.

Note that method **EvaluateUpdate** is a recursive one which returns a relation. The returned relation is used to replace the one be updated.

Because of the surrogate reuse policy, it is necessary to use new surrogate and duplicate copy of relations for the nested attributes involved in update.

4.4.4 View

A View is a computed relation defined on relations, views, and computations. It is evaluated to a relation when it is referred to in expressions other than those for view declarations. When a view is declared, if there exists a relation of the same name in the system table, simply set the view flag and insert the syntax tree of the view. Otherwise a new entry is created and inserted into the system table. Note that the order of declaration of relation and view of the same name matters. If the

```
domain A, B, C integer;
domain X(B, C);
relation R(A, X) <− {(3,{(2,2),(3,3),(4,4)}), (6,{(4,4),(5,5)})};
update R change
A <− 2*A,
update X change
B <− C,
C <− 2 + C
using ijoin where C = 2 in X
using ijoin where A <= 3 in R;
```

Figure 4.15: An example of nested update.

EvaluateUpdate(node)

rel ← part of relation to be updated
for each statement S in after keyword `change`
    case ASSIGNMENT of the form : *Identifier* ← *Expression*
        actualize *Expression* on rel
    case UPDATE of the form : `update` *Identifier* `change` ...
        replace the update statement with the following assignment
        Identifier ← `EvaluateUpdate`(S.node)
complete the update using the unnested update

Figure 4.16: Algorithm for nested update.

declaration of view comes after that of the relation of the same name, the information on the relation definition is retained to provide initial value for the view if the view is recursive. If the declaration of relation comes after that of the view of the same name, the view definition is completely replaced by the relation. In our implementation, the semantic checks are delayed to runtime, i.e., there is no check for the existence of the relations, views and computations on which the view is defined at the declaration time. The view definition is made permanent by store its expression tree on disk using the object serialization machenism provided in Java API.

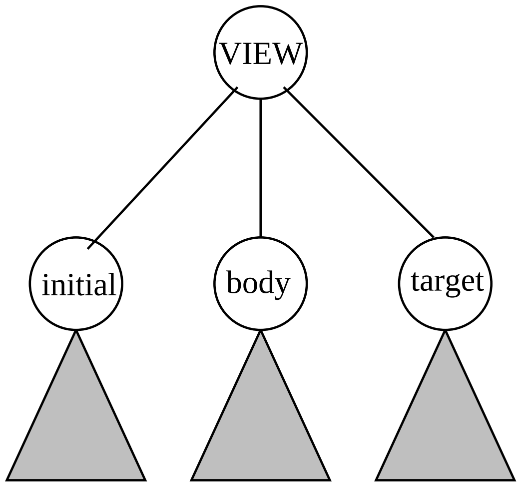


Figure 4.17: Syntax tree of view. Note that *initial*, *body* and *target* are roots of three trees of expressions. The *initial* and *target* parts are optional.

A view is recursive if its definition refers to itself directly or indirectly through other views. For evaluating non-recursive view, simply call the method **EvaluateExpression** with the syntax tree of the view body as argument. For recursive view [AU79, BR86, CLM89], we build a directed graph in which nodes represent names of relations and views, and edges represent dependence between them, and then look for the strongly connected components in the graph. After the strongly connected components are found, the recursive views are placed in loops, one for each strongly connected component. These views are properly initialized (using the results of *initial* expressions or the relations of the same name as the views if no there is no *initial* part, or the empty relations otherwise) and the loop is iterated until all of the relations reach their targets (the results of the *target* expressions) or stop changing.

Figure 4.18 shows the definition of two mutual views VB and VC, and the directed graph associated with the definition.

domain A, B **integer**;

relation R(A, B) <− {(1,1),(5,5)};

relation S(A, B) <− {((2,2),(3,3),(4,4),(5,5)};

VB **initial** R **is** VB **ujoin** VC **ujoin** **pick** S;

relation VC(A, B);

VC **is** VC **ujoin** **pick** (S **djoin** VB);

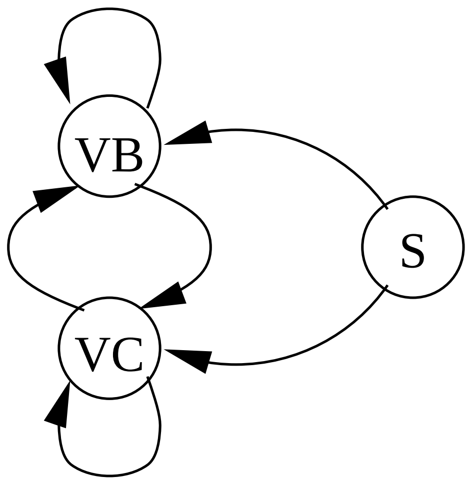


Figure 4.18: Two mutual recursive views VB and VC, and the associated directed graph. Note that in the directed graph, the edge from S to VB indicates that the definition of VB depends on S. There is one strongly connected component that contains VB and VC, so VB and VC are mutual recursive views.

In our implementation, we integrate the semantic check for the existence of all relations and views being referenced into the graph building process. Therefore the evaluation process can always proceed safely.

4.5 Performance

In this section, we summarize some performance test results. All the tests are run on Pentium-MMX200 (32M RAM) running Linux Operating System version 2.0. The time is clock time and in unit of second.

4.5.1 Tests on Flat Relations

The tests in this section are performed on three flat relations R, S, and T. The attribute values are generated randomly. Figure 4.19 shows the definition of these relations. The binary files for relations R, S, and T have 120000, 80000 and 80000 bytes respectively.

`domain` A, B, C `integer`;

`relation` R (A, B, C) <– {10,000 tuples};

`relation` S (A, B) <– {10,000 tuples};

`relation` T (B, C) <– {10,000 tuples};

Figure 4.19: Three flat test relations R, S, and T. Relation R is defined on three `integer` attributes; relation S and T are defined on two `integer` attributes. All of them have 10,000 randomly generated tuples.

Tests for initialization of relation, assignment, projection, selection, join, horizontal domain operation, and vertical domain operation. The test conditions for each test is in the following. Note that relation R is sort on attributes A, B, and C after initialization. So there is no sort in test 1 and 3. The time used for each test is summarized in table 4.2.

- Test0 – Initialization of relation R. The time is that used to interpret the initialization tree. The time used to parse the input into initialization tree is not included.

- Test1 – Assignment without sort: Temp [A, B <– A, B] R;

- Test2 – Assignment with sort: Temp [B, C <– C, B] R;
- Test3 – Project on sorted attribute: Temp <– [A] `in` R;
- Test4 – Project on unsorted attribute: Temp <– [B] `in` R;
- Test5 – Selection: Temp <– `where` B > 500 `in` R;
- Test6 – Join two relations using `ijoin`: Temp <– S `ijoin` T;
- Test7 – Join two relations using `ujoin`: Temp <– S `ujoin` T;
- Test8 – Horizontal operation: `let` X `be` A + B + C; Temp <– [A, B, C, X] `in` R;
- Test9 – Vertical operation: `let` Y `be` `red` + `of` A; Temp <– [A, B, C, Y] `in` R;

Test	Total Time	Actualize Time
Test0	3.018	
Test1	0.722	
Test2	2.697	
Test3	1.567	
Test4	3.509	
Test5	2.745	
Test6	3.542	
Test7	4.435	
Test8	5.648	0.250
Test9	5.895	0.181

Table 4.2: Time for tests on the flat relations R, S, and T. The Actualize Time is the CPU time used to actualize virtual attributes (X in Test8 and Y in Test9).

The time to load relation R from disk is about 0.4 seconds. The time to save relation R to disk is also about 0.4 seconds. From test 1 and 2 (or test 3 and 4), we deduce that the time used to sort relation R is 1.975 seconds.

4.5.2 Tests on a Nested Relation

The tests in this section are performed on the nested relation R. Relation R is defined on two nested attributes S and T. Figure 4.20 shows the definition of relation R. There are 1000 randomly generated tuples in relation R. Each nested attribute value has an average of 5 tuples. The binary files for relations R, .S, and .T have 160000, 81008 and 81392 bytes respectively. The tuple-by-tuple approach [Yua98] is used for the domain algebra operations on nested attributes. The time used for each test is summarized in table 4.3.

domain A, B **integer**;
domain S, T (A, B);
relation R (S, T) <− {1000 tuples, each nested attribute value has an average of 5 tuples};

Figure 4.20: A nested test relation R defined on nested attributes S and T. R has 1000 randomly generated tuples. Each nested attribute value has an average of 5 tuples.

- Test0 – Initialization of relation R. The time is used to interpret the initialization tree. The time used to parse the input into initialization tree is not included.
- Test1 – Project on unsorted attribute: Temp <− [B] **in** R;
- Test2 – Horizontal `ijoin`: **let** X **be** S **ijoin** T; Temp <− [S, T, X] **in** R;
- Test3 – Horizontal `ujoin`: **let** Y **be** S **ujoin** T; Temp <− [S, T, Y] **in** R;

Test	Total Time	Actualize Time
Test0	5.249	
Test1	52.322	
Test2	63.231	7.728
Test3	77.242	21.093

Table 4.3: Time for tests on the nested relation R. The Actualize Time is the CPU time used to actualize virtual attributes (X in Test2 and Y in Test3).

The time to load relation R and two *dot relations* .S and .T from disk is about 0.65 seconds. The time to save these relations to disk is about 0.7 seconds. From the test results, we find that sort relation R on nested attributes uses about 50 seconds.

4.5.3 Test on a Recursive View

For the recursive view defined in figure 4.21, the evaluation of view V starts with 100 tuples, and iterates 101 times until there is no tuple in it. The total time used is 3.427 seconds.

```
domain A, B, C integer;
relation R (A, B, C) <- {100 tuples};
V initial R is V djoin pick V;
Temp <- V;
```

Figure 4.21: Test of a recursive view. In the evaluation process, view V starts with 100 tuples, and iterates 101 times until there is no tuple in it.

From the above tests, we can also conclude that the time used for data manipulation in CPU and RAM exceeds the time for disk IO for large relations. This agrees with the former Relix implementation.

Chapter 5

Conclusions

In this chapter, we summarize the work done in this thesis, and propose the work to do in the future.

5.1 Summary

In this thesis, we describe the redesign of the former Relix and the Java implementation to integrate the notion of nested relations into both the relational algebra and the domain algebra. In our implementation, we use the syntax of former Relix wherever it is possible. The integration of nested relations into the flat relations is accomplished through the following mechanisms.

- We introduce some new syntax which is special for nested relations. The new syntaxes includes the declaration of nested attributes and relations, the initialization of nested relations, and the nested update.

- We integrate the relational expressions into domain expressions so that relational operations can be applied to nested attributes. This integration allows the application of domain algebra operations to nested attributes at any level of the nesting hierarchy.

- We give definition of order between relations so that relation-valued attributes are treated the same way as attributes of primitive type. The order is defined in a way that an efficient algorithm for finding the order between relations is possible. In our definition, the order depends on the structure of the nested relations, i.e., the way the relations are nested.

- We use a set of flat relations to model a nested relation. These flat relations are linked to each other by surrogates.

With the above mechanisms, our implementation provides a uniform syntax and notation for both flat and nested relations, and supports relational algebra for nested relations with arbitrary level of nesting, and domain algebra for nested attributes at any level of the nesting hierarchy.

The whole implementation is written entirely in Java to take advantage of Java's platform independence. Tests for performance are done and special attentions are paid to efficiency to overcome Java's low performance. The resulting system achieved platform independence for both the Java code and the binary data files for relations: the code runs successfully on many platforms including Window 95, Window NT, Linux, and Solaris; the underlying data for databases are also reused across these platforms. Performance results also show that our implementation runs reasonably fast compared with the former Relix which is implemented in the C programming language.

5.2 Future Work

Due to the time limit, we implemented only a large subset of the entire syntax described in Appendix B. To give full support of the entire syntax, we need to add the following features.

- The σ-join (except for `icomp` or `natcomp`) which includes `eqjoin` (`gtjoin`), `gejoin` (`sup`, `div`), `ltjoin`, `lejoin` (`sub`), and `iejoin` (`sep`).

- The support for quantifiers in selection.

- The concurrent access control, i.e., the `when ... in` (`from`) expression, and the sequential ($--$), parallel ($||$), and choice ($??$) operators for statements.

The support for σ-join operators also enriches the conditional expression in selection, besides their ordinary use in relational expressions. For example, we may answer the following query using operator `gejoin` (`sup`, `div`).

`Query`
Find the department for which professor Franklin Wong (with salary 48) works.
`relation` ProfTmp(name, salary) $<-$ {("Franklin Wong", 48)};
`pr where` professor `sup` ProfTmp `in` Management;

There are still more features to be added to refine the implementation.

- Because we reuse the surrogates without using reference counters for them, in some *dot relations*, it is possible to have data items which are not referenced by any relation. These unreferenced data items not only consume disk space, but also reduce the performance of the system. They should be garbage collected by the system. The garbage collection can be performed using the garbage collection algorithm for Java heap: each surrogate in all the *dot relations* is checked to see if it is referenced by other relations; those data items whose surrogates are not referenced are deleted from their *dot relations*.

- A graphical user interface (GUI) that represents the structure and all the data of a nested relation and allows direct manipulation of attribute values for update is desirable. The current implementation that displays a nested relation through a set of flat relations linked by surrogates is not intuitive and proves hard to understand by the user.

- Recursively nested relations should be supported by allowing recursive definition of attributes.

Appendix A

Backus-Naur Form for the Parser

This appendix shows the Backus-Naur form (BNF) of the grammar in our implementation. The convention of the BNF definition is shown in table A.1.

Form	Meaning
<SYMBOL>	SYMBOL is a definition of token and must be substituted
"SYMBOL"	SYMBOL is reserved word or symbol and must be typed as it is
S1 | S2	either S1 or S2 can be used
(SYMBOL)?	SYMBOL is optional
(SYMBOL)*	SYMBOL may appear zero or more times
(SYMBOLS)	grouping SYMBOLS as one unit for high precedence

Table A.1: BNF convention.

The grammar is created from the grammar specification (in file Parser.jjt), using the JavaCC documentation generator called `jjdoc`. Because JavaCC is a top-down parser, left-recursion is not allowed in the grammar specification. Therefore the grammar looks different from that of former Relix which is intended for the bottom-up parser generator Yacc.

There are five token definitions: <EOF> for end-of-file; <IDENTIFIER> for identifier; <INTEGER_LITERAL> for integer constants; <FLOAT_LITERAL> for floating constants; and <STRING_LITERAL> for string constants. The formal definitions of identifier and constants are given in section 3.2.

```
Start := Command ";"
    | Statement ";"
    | ";"
    | <EOF>

Command := "help" (<IDENTIFIER>)?
    | "quit"
    | "input" FilePath
    | "debug"
    | "batch"
    | "expert"
    | "time"
    | "deld" IDList
    | "delr" IDList
    | "pr" Expression
    | "sd" (<IDENTIFIER>)?
    | "sr" (<IDENTIFIER>)?
    | "srd"
    | "ssd"
    | "ssr"
    | "print" <STRING_LITERAL>

Statement := SequentialStatement

SequentialStatement := ParallelStatement ("--" ParallelStatement)*

ParallelStatement := ChoiceStatement ("||" ChoiceStatement)*

ChoiceStatement := PrimaryStatement ("??" PrimaryStatement)*

PrimaryStatement := Declaration
    | Assignment
    | Update
    | ComputationCall
    | Conditional
    | ForLoop
    | WhileLoop
    | Exit
    | DeadLock
    | Exec
    | StatementBlock
```

```
StatementBlock := "{" Statement (";" Statement)* (";")? "}"

Conditional := "if" Expression "then" Statement ("else" Statement)?

ForLoop := ("for" Identifier)? ("from" Expression)?
           ("to" Expression)? ("by" Expression)?
           ("do" | "loop") Statement

WhileLoop := "while" Expression ("do" | "loop") Statement

Exit := "exit"

DeadLock := "deadlock"

Exec := "exec" Identifier

Declaration := "relation" IDList "(" IDList ")" (Initialization)?
    | Identifier ("initial" Expression)? "is" Expression
      ("target" Expression)?
    | "domain" IDList Type
    | "let" Identifier ("initial" Expression)? "be" Expression
    | ("computation" | "comp") Identifier
      "(" (ParameterList)? ")" "is" ComputationBody

Initialization := "<-" ("{" ConstantTupleList "}" | Identifier)

ConstantTupleList := ConstantTuple ("," ConstantTuple)*

ConstantTuple := "(" Constant ("," Constant)* ")"

Constant := Literal
    | "{" ConstantTupleList "}"

Identifier := <IDENTIFIER>

FilePath := <STRING_LITERAL>

Assignment := Identifier
    ( ("<-" | "<+") Expression
      |
      "[" IDList ("<-" | "<+") ExpressionList "]" Expression
    )
```

```
Update := "update" Identifier
    ( ("add" | "delete") Expression
      |
      "change" (StatementList)? (UsingClause)?
      |
      "[" IDList ("add" | "delete") ExpressionList "]" Expression
    )

StatementList := Statement ("," Statement)*

UsingClause := "using"
    ( JoinOperator Expression
      |
      "[" ExpressionList ":" JoinOperator (":")?
      ExpressionList "]" Expression
    )

IDList := Identifier ("," Identifier)*

ExpressionList := Expression ("," Expression)*

Expression := Disjunction

Disjunction := Conjunction (("or" | "|") Conjunction)*

Conjunction := Comparison (("and" | "&") Comparison)*

Comparison := Concatenation (ComparativeOperator Concatenation)?

Concatenation := MinMax ("cat" MinMax)*

MinMax := Summation (("min" | "max") Summation)*

Summation := JoinExpression (("+" | "-") JoinExpression)*

JoinExpression := Projection
    ( JoinOperator Projection
      |
      "[" ExpressionList ":" JoinOperator (":")?
      ExpressionList "]" Projection
    )*

Projection := Projector (("in" | "from") Projection | Selection)
    | Selection
```

```
Projector := (QuantifierOperator)? "[" (ExpressionList)? "]"

Selection := Selector
    | QSelector
    | Term

Selector := ("where" | "when") Expression ("in" | "from") Projection
    | "edit" (Projection)?
    | "zorder" Projection

QSelector := "quant" QuantifierList (("where" | "when") Expression)?
             ("in" | "from") Projection

QuantifierOperator := "." | "%" | "#"

QuantifierList := Quantifier ("," Quantifier)*

Quantifier := "(" Expression ")" Expression

Term := Factor (("*" | "/" | "mod") Factor)*

Factor := ("+" | "-" | "not" | "!") Factor
    | Power

Power := Primary ("**" Power)*

Primary := Literal
    | QuantifierOperator
    | ArrayElement
    | PositionalRename
    | Identifier
    | Cast
    | "(" Expression ")"
    | Pick
    | Eval
    | Function
    | IfThenElseExpression
    | VerticalExpression

ArrayElement := Identifier "[" ArrayIndexList "]"

ArrayIndexList := (Expression)? ("," (Expression)?)*
```

```
PositionalRename := Identifier "(" (IDList)? ")"

Cast := "(" Type ")" Primary

Pick := "pick" Selection

Eval := "eval" Expression

Function := FunctionOperator "(" Expression ")"

Literal := "null"
    | "dc"
    | "dk"
    | "true"
    | "false"
    | ("+" | "-")? (<INTEGER_LITERAL> | <FLOAT_LITERAL>)
    | <STRING_LITERAL>

IfThenElseExpression := "if" Expression "then" Expression
                        "else" Expression

VerticalExpression := "red" AssoCommuOperator "of" Expression
    | "equiv" AssoCommuOperator "of" Expression
      "by" ExpressionList
    | "fun" OrderedOperator "of" Expression
      "order" ExpressionList
    | "par" OrderedOperator "of" Expression
      ( "order" ExpressionList "by" ExpressionList
        | "by" ExpressionList "order" ExpressionList
      )

Type := ("boolean" | "bool")
    | "short"
    | ("integer" | "intg")
    | "long"
    | ("float" | "real")
    | "double"
    | ("string" | "strg")
    | "text"
    | ("statement" | "stmt")
    | ("expression" | "expr")
    | ("computation" | "comp") "(" IDList ")"
    | "(" IDList ")"
```

```
AssoCommuOperator := ("or" | "|")
    | ("and" | "&")
    | "min"
    | "max"
    | "+"
    | "*"
    | ("ijoin" | "natjoin")
    | "ujoin"
    | "sjoin"

OrderedOperator := AssoCommuOperator
    | "cat"
    | "-"
    | "/"
    | "mod"
    | "**"
    | "pred"
    | "succ"

ComparativeOperator := "substr"
    | "="
    | "!="
    | ">"
    | "<"
    | ">="
    | "<="

JoinOperator := "nop"
    | MuJoin
    | (("not" | "!"))? SigmaJoin

MuJoin := ("ijoin" | "natjoin")
    | "ujoin"
    | "sjoin"
    | "ljoin"
    | "rjoin"
    | ("dljoin" | "djoin")
    | "drjoin"

SigmaJoin := ("icomp" | "natcomp")
    | "eqjoin"
    | ("gejoin" | "sup" | "div")
    | "ltjoin"
    | ("lejoin" | "sub")
```

```
    | ("iejoin" | "sep")

FunctionOperator := "abs"
    | "sqrt"
    | "sin"
    | "asin"
    | "cos"
    | "acos"
    | "tan"
    | "atan"
    | "sinh"
    | "cosh"
    | "tanh"
    | "log"
    | "log10"
    | "round"
    | "ceil"
    | "floor"
    | "isknown"
    | "chr"
    | "ord"

ParameterList := Parameter ("," Parameter)*

Parameter := <IDENTIFIER> (":" "seq")?

ComputationBody := ComputationDeclarationAndInitialization
                   ComputationBlock ("alt" ComputationBlock)*

ComputationBlock := "{" ComputationStatements "}"

ComputationDeclarationAndInitialization :=
    ( LocalVariableDeclaration
      |
      StateVariableDeclaration
      |
      ComputationInitialization
    )*

LocalVariableDeclaration := "local" IDList Type
                            (VariableInitialization)? ";"

StateVariableDeclaration := "state" IDList Type
                            (VariableInitialization)? ";"
```

```
ComputationInitialization := IDList "<-" Expression ";"

VariableInitialization := "<-" Expression

ComputationStatements := Statement (";" Statement |
                         "also" Statement)* (";")?

ComputationCall := Identifier "(" (CallParameterList)? ")"

CallParameterList := CallParameter ("," CallParameter)*

CallParameter := ("in" | "out") <IDENTIFIER>
```

Appendix B

Attribute Domains and Storage Format

This appendix describes the representations of the attribute values in primary memory (RAM), domains of primitive attribute types, and the storage format of relations.

Our implementation supports the primitive attribute types including `boolean`, `short`, `integer`, `long`, `float` and `double`. Other attribute types can be represented by these primitive attribute types. For example, type `expression` and `statement` are internally represented by type `string`. Table B.1 shows the Java representation in RAM and the size and format for each of these attribute types.

Each of the primitive attribute types supports two null values, `dk` for don't know and `dc` for don't care. We use Java type byte to represent attribute type `boolean`, because attribute type `boolean` includes `dk` and `dc`, besides `true` and `false`. Two special values in each type are reserved for representing `dk` and `dc`. Table B.2 shows the values which are used for `dk` and `dc`. The values for `dk` and `dc` are chosed in order to follow the former Relix conventions, i.e., `dk` is less than `dc` while `dc` is less than any other value except `dk` and `dc`. Note that for attribute type `string` and `text`, `dk` and `dc` start with the smallest charactor "\u0000". Values such as Short.MIN_VALUE, Integer.MIN_VALUE, and Float.NEGATIVE_INFINITY are defined in the Java API

Attribute Type	Java Type In RAM	Size and Format
`boolean`	byte	1 byte
`short`	short	2 bytes unsigned
`integer`	int	4 bytes unsigned
`long`	long	8 bytes unsigned
`float`	float	4 bytes IEEE 754
`double`	double	8 bytes IEEE 754
`string`	String	variable length
`text`	String	variable length

Table B.1: Primitive attribute types and their in RAM representations.

package java.lang.

The domains of the primitive attribute types are those for the Java types used to represent them, except for the values reserved for `dk` and `dc`. The only one exception is that `true` and `false` in type `boolean` are represented by 1 and 0 in Java byte type, and all other values except -2 for `dk` and -1 for `dc` are invalid.

Attribute Type	Value for `dk`	Value for `dc`
`boolean`	-2	-1
`short`	Short.MIN_VALUE	Short.MIN_VALUE + 1
`integer`	Integer.MIN_VALUE	Integer.MIN_VALUE + 1
`long`	Long.MIN_VALUE	Long.MIN_VALUE + 1
`float`	Float.NEGATIVE_INFINITY	- Float.MAX_VALUE
`double`	Double.NEGATIVE_INFINITY	- Double.MAX_VALUE
`string`	“\u0000dk”	“\u0000dc”
`text`	“\u0000dk”	“\u0000dc”

Table B.2: Reserved values for `dk` and `dc` for each primitive attribute types.

We use binary storage format for relations. The big advantage of the binary format is that it saves disk space and I/O time. Platform independence of data files is also achieved.

A relation is stored in a disk file in the tuple-by-tuple fashion, while a tuple is stored in the attribute-by-attribute fashion. Each attribute value is stored according to the following rules.

- For attribute of type `boolean`, `short`, `integer`, `long`, `float`, or `double`, the Java representation of the attribute value is stored according to the rules specified in the Java language specification. Surrogates are the same as `long` when they are stored on disk.

- For attribute of type `string`, a short integer followed by a Java byte array representing the attribute value is stored. The short integer is used to indicate the length of the string according to the following rule.

  $$\text{value of the short integer} = \text{length of the string} + \text{Short.MIN\_VALUE} + 2$$

  Therefore, the attribute type `string` supports character string up to 65533 in length[1].

  If the attribute value is `dk`, only the short integer value Short.MIN_VALUE is stored. If the attribute value is `dc`, only the short integer value Short.MIN_VALUE + 1 is stored.

- For attribute of type `text`, an integer followed by a Java byte array representing the attribute value is stored. The integer is used to indicate the length of the string according to the following rule.

  $$\text{value of the integer} = \text{length of the text} + \text{Integer.MIN\_VALUE} + 2$$

  Therefore, the attribute type `text` supports character string up to 4294967293 in length[2]. If the attribute value is `dk`, only the integer value Integer.MIN_VALUE

[1] Short.MAX_VALUE − Short.MIN_VALUE − 2 = 65533

[2] Integer.MAX_VALUE − Integer.MIN_VALUE − 2 = 4294967293

is stored. If the attribute value is `dc`, only the integer value Integer.MIN_VALUE + 1 is stored.

Because the way the values of primitive Java data types are stored on disk is platform independent, the data files for relations stored according to the above rules are platform independent.

Bibliography

[AFS89] S. Abiteboul, P. C. Fischer, and H. J. Schek. *Nested Relations and Complex Objects in Databases.* Lecture Notes in Computer Science 361. Springer-Verlag, Berlin, 1989.

[AG96] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[AMM83] H. Arisawa, K. Moriya, and T. Miura. Operations and the properties on non-first-normal-form relational databases. In *Proceedings of the 9th International Conference on Very Large Data Bases*, pages 197–204, 1983.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, 1986.

[AU79] A. V. Aho and J. D. Ullman. Universality of data retrieval languages. In *Sixth Annual Symposium on Principles of Programming Languages*, pages 110–117, 1979.

[Bak98] Patrick Baker. Java implementation of computations in a database programming language. Master's thesis, McGill University, to appear, 1998.

[BR86] F. Bancilhon and R. Ramakrishnan. An amateur's introduction to recursive query processing strategies. In *Proceedings on ACM SIGMOD*, pages 16–52, 1986.

[CLM89] A. Clouâtre, N. Laliberté, and T. H. Merrett. A general implementation of relational recursion with speedup techniques for programmers. Technical Report TR-SOCS-89.6, School of Computer Science, McGill University, 1989.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithm.* The MIT Press, McGraw-Hill Book Company, 1990.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[Cod79] E. F. Codd. Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4:397–434, 1979.

[DG88] A. Deshpande and D. Van Gucht. An implementation of nested relational database. In *Proceedings of the 14th International Conference on Very Large Data Bases*, pages 266–274, 1988.

[Dou91] Samir Dourik. Implementation of delay and nondeterminism in a database programming language. Master's thesis, McGill University, Montreal, 1991.

[FG85] P. C. Fisher and D. Van Gucht. Determining when a structure is a nested relation. In *Proceedings of the 11th International Conference on Very Large Data Bases*, pages 171–180, 1985.

[FT83] P. C. Fischer and S. J. Thomas. Operators for non-first-normal-form relations. In *Proceedings of IEEE Computer Software and Application Conference*, pages 464–475, 1983.

[GJS96] James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[He97] Hongbo He. Implementation of nested relations in a database programming language. Master's thesis, McGill University, Montreal, 1997.

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

[JS82] G. Jaeschke and H. J. Schek. Remarks on the algebra of non-first-normal-form relations. In *Proceedings of the First ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 124–138, Los Angeles, 1982.

[Kom92] Maria Komioti. Implementation of process synchronization in a database programming language. Master's thesis, McGill University, Montreal, 1992.

[KR89] H. F. Korth and M. A. Roth. Query languages for nested relational databases. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[Lal86] N. Laliberté. Design and implementation of a primary memory version of aldat. Master's thesis, McGill University, Montreal, 1986.

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

[LS88] R. Lorie and H. J. Schek. On dynamically defined objects and SQL. In *Proceedings of 2nd Workshop on Object-Oriented Database Systems*, Bad Münster, 1988.

[Mak77] A. Makinouchi. A consideration on normal form of not-necessarily-normalized relation in the relational data model. In *Proceedings of 3rd International Conference on Very Large Data Bases*, pages 447–453, Tokyo, 1977.

[Mer76] T. H. Merrett. MRDS: An algebraic relational database system. In *Canadian Computer Conference*, pages 102–124, Montreal, 1976.

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

[Mer84] T.H. Merrett. *Relational Information Systems.* Reston Publishing Co., Reston, VA, 1984.

[ML89] T.H. Merrett and N. Laliberté. Including scalars in a programming language based on the relational algebra. Technical Report TR-SOCS-89.5, School of Computer Science, McGill University, June 1989.

[MLG+88] T.H. Merrett, N. Laliberté, B. Gunnlaugsson, M. Tsakalis, and A. Chong. Relix - first steps to an operating language. Technical Report TR-SOCS-88.1, School of Computer Science, McGill University, January 1988.

[OY87] Z. M. Ozsoyoglu and L. Y. Yuan. A new normal form for nested relations. *ACM Transactions on Database Systems*, 12(1):111–136, 1987.

[PA86] P. Pistor and F. Anderson. Designing a generalized NF^2 model with an SQL-type language interface. In *Proceedings of the 12th International Conference on Very Large Data Bases*, pages 278–285, 1986.

[SAB+89] M. H. Scholl, S. Abiteboul, F. Bancilhon, N. Bidoit, S. Gamerman, D. Plateau, P. Richard, and A. Verroust. VERSO: A database machine based on nested relations. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[SDV] Sriram Sankar, Rob Duncan, and Sreenivasa Viswanadha. Java Compiler Compiler (JavaCC)-The Java Parser Generator. JavaCC web site at: http://www.suntest.com/JavaCC/. The web site contains documentations, FAQs, newsgroups, and softwares for JavaCC and JJTree.

[SS87] M. H. Scholl and H. J. Schek. Theory and applications of nested relations and complex objects. In *An International Workshop: Workshop Material*, Darmstadt, West Germany, 1987.

[Tak89] K. Takeda. On the uniqueness of nested relations. In *Nested Relations and Complex Objects in Databases, Lecture Notes in Computer Science 361*. Springer-Verlag, Berlin, 1989.

[TF86] S. J. Thomas and P. C. Fischer. Nested relational structures. In P. C. Kanellakis, editor, *Advances in Computing Research III, The Theory of Databases*, pages 269–307. JAI Press, 1986.

[Yua98] Zhongxia Yuan. Java implementation of the nested domain algebra in a database programming language. Master's thesis, McGill University, to appear, 1998.