INFORMATION TO USERS

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

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

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

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

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

UMI

Bell & Howell Information and Learning
300 North Zeeb Road, Ann Arbor, MI 48106-1346 USA
800-521-0600
TOWARDS AN IMPLEMENTATION OF SchemaLog – A DATABASE PROGRAMMING LANGUAGE

ALANOLO Y JOSEPH ANDREWS

A THESIS
IN
THE DEPARTMENT
OF
COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

DECEMBER 1997
© ALANOLO Y JOSEPH ANDREWS, 1998
The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author’s permission.

L’auteur a accordé une licence non-exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L’auteur conserve la propriété du droit d’auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-39984-2
Abstract

Towards An Implementation of SchemaLog – A Database Programming Language

Alanoly Joseph Andrews

The efficient implementation of advanced database programming languages calls for investigating novel architectures and algorithms. In this thesis, we discuss our implementation of SchemaLog, a logic-based database programming language, capable of offering a powerful platform for a variety of database applications involving data/meta-data querying and restructuring. Our architecture for the implementation is based on compiling SchemaLog constructs into an extended version of relational algebra called Schema Algebra (SA). We modify the new operators in this algebra to suit our implementation, and we illustrate how a SchemaLog program can be evaluated by conversion into an expression in SA. Based on this algebra, we develop a top-down algorithm, using the Rule/Goal Tree method, for evaluating SchemaLog programs. We then discuss three alternative storage structures for the implementation and study their effect on the efficiency of implementation. For each storage structure, we propose strategies for implementing our algebraic operators. We have implemented all these algorithms, using MicroSoft Access DBMS running on Windows 3.1, and have run an extensive set of experiments for evaluating the efficiency of alternative strategies under a varied mix of querying and restructuring operations and varying parameters on the type and size of data. We discuss the results of our experiments and make recommendations on the type of storage structures to be used.
Acknowledgments

It should come as no surprise that I start by acknowledging my debt of gratitude to Prof. Laks V.S. Lakshmanan, my thesis advisor and supervisor. He has played a major role throughout the work I have done towards this presentation of my thesis and has seen to it that it has finally seen the light of day. His erudition in the area of logic and databases is well-known. The long list of publications in the subject in the last few years bears ample testimony to that assertion. It is my hope that some of that expertise, as well as enthusiasm for continued research has rubbed off on his students, including me.

My work in this thesis is based in part on the work done by Iyer Subramanian, alias Subbu, for his doctoral dissertation. During the course of my work of implementation, I naturally had to spend long hours with Subbu to share ideas on such “inane” topics as data structures, algorithms, etc. (not to mention the more interesting hours we had discussing more “pagan” subjects). I am grateful to Subbu for the valuable contributions he has made to making this thesis a reality.

Another fellow-student with whom I have had close contact through the years in Concordia is Nematollaah Shiri. Though not directly involved with my area of research, Shiri was a constant presence, a friend to whom I could turn in any circumstance.

Financial support for this thesis was provided in part by the Natural Sciences and Engineering Research Council of Canada and the Fonds Pour Formation De Chercheurs Et L'Aide À La Recherche of Quebec.

I would also like to thank the support staff in the Computer Science Department: the analysts (Stan Swiercz deserves special mention here), and the secretaries (in particular, Edwina, Halina and Stephanie) for their quick response to questions and problems.

Finally, I would like to thank the members of my own family (my wife Mary, and
my children Deepti, Preeti and Jyoti) for their love and support for me during some difficult years. The early years of my stay in Montreal were spent with my brother Jose. I would like to express here my gratitude to him, his wife Ruby and their children. Thanks also to my brother James who, too, was a pillar of support during the past years.
# Contents

List of Figures ix

List of Tables xi

1 Introduction 1
   1.1 Opening Remarks 1
   1.2 Outline of the thesis 3
   1.3 Contributions of this thesis 4

2 Basics of SchemaLog and Schema Algebra 6
   2.1 Introduction 6
   2.2 Syntax of SchemaLog 6
   2.3 Operators in Schema Algebra 9
      2.3.1 Definitions of Operators 10
   2.4 Further Examples 17
   2.5 Concluding Remarks 21

3 Top-Down Processing of SchemaLog Programs 24
   3.1 Introduction 24
   3.2 Atomizing a SchemaLog program 25
   3.3 Outline of the Top-Down Processing procedure 26
   3.4 Top-Down Algorithm for SchemaLog 28
   3.5 Example of Rule-Goal Tree Expansion in SchemaLog 31
   3.6 Conclusion 33

4 Physical Storage Architectures 34
   4.1 Introduction 34
4.2 Conventional Storage ........................................ 35
  4.2.1 Selection .................................................. 35
  4.2.2 Projection ............................................... 37
  4.2.3 Join ...................................................... 38
  4.2.4 Fetching Relation Names ................................. 40
  4.2.5 Fetching Relations and their Schemas .................. 40
  4.2.6 Querying Data and Meta-data .......................... 40
  4.2.7 Creating Relations .................................... 42
  4.2.8 Creating Relations with Schemas ...................... 42
  4.2.9 Creating Relations with Schema and Data ............. 43

4.3 Reduced Storage ............................................ 44
  4.3.1 Selection ................................................ 45
  4.3.2 Projection ............................................. 46
  4.3.3 Join ..................................................... 47
  4.3.4 Fetching Relation Names ................................. 48
  4.3.5 Fetching Relations and their Schemas .................. 48
  4.3.6 Querying Data and Meta-data .......................... 49
  4.3.7 Creating Relations .................................... 51
  4.3.8 Creating Relations with Schemas ...................... 51
  4.3.9 Creating Relations with Schema and Data ............. 51

4.4 Reduced, Atomized Storage ................................. 52
  4.4.1 Selection ................................................ 54
  4.4.2 Projection ............................................. 54
  4.4.3 Join ..................................................... 55
  4.4.4 Fetching Relation Names ................................. 56
  4.4.5 Fetching Relations and their Schemas .................. 56
  4.4.6 Querying Data and Meta-data .......................... 56
  4.4.7 Creating Relations .................................... 58
  4.4.8 Creating Relations with Schemas ...................... 59
  4.4.9 Creating Relations with Schema and Data ............. 59

4.5 Conclusion .................................................. 60

5 Experimental Results ....................................... 62
  5.1 Introduction ............................................... 62
5.2 Preparation of the Test Bed ........................................... 62
5.3 Implementation of Schema Algebra operators ..................... 63
   5.3.1 Experimental Results and Graphs .............................. 64
5.4 Analysis of Results ....................................................... 81
   5.4.1 Individual Operations ............................................. 81
   5.4.2 Mix of Operations ............................................... 82
   5.4.3 Effect of table size on cost of $SA$ operations ............... 83
5.5 Conclusion ................................................................. 87

6 Conclusion ........................................................................ 88
   6.1 Comparison ................................................................... 88
   6.2 Summary ................................................................. 90
   6.3 Future Work ............................................................. 91

Bibliography ....................................................................... 92
# List of Figures

2.1 Example Illustrating Database Restructuring ........................................... 9  
2.2 The NYSE Database ................................................................................. 11  
2.3 Example for creating relation schemas .................................................... 15  
2.4 Example for creating and populating relations .......................................... 16  
2.5 $r_1$: the result of Step 1 ......................................................................... 19  
2.6 $r_2$: the result of Step 2 .......................................................................... 20  
2.7 $r_3$: the result of Step 3 .......................................................................... 20  
2.8 $r_4$: the result of Step 4 .......................................................................... 20  
2.9 $r_5$: the result of Step 5 .......................................................................... 21  
2.10 The result of Step 6 ................................................................................ 22  
3.1 Example of Rule/Goal Tree ...................................................................... 33  
4.1 Conventional Storage Tables .................................................................... 45  
4.2 Reduced Storage Tables ........................................................................... 46  
4.3 Reduced, Atomized Storage Tables ........................................................... 53  
5.1 Individual Operations (from Table 5.1) ..................................................... 66  
5.2 Mix of Operations (from Table 5.1) ......................................................... 67  
5.3 Individual Operations (from Table 5.2) ..................................................... 68  
5.4 Mix of Operations (from Table 5.2) ......................................................... 68  
5.5 Individual Operations (from Table 5.3) ..................................................... 69  
5.6 Mix of Operations (from Table 5.3) ......................................................... 70  
5.7 Individual Operations (from Table 5.4) ..................................................... 71  
5.8 Mix of Operations (from Table 5.4) ......................................................... 71  
5.9 Individual Operations (from Table 5.5) ..................................................... 72  
5.10 Mix of Operations (from Table 5.5) ......................................................... 73  
5.11 Individual Operations (from Table 5.6) ..................................................... 75  
5.12 Mix of Operations (from Table 5.6) ......................................................... 75
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Execution time with table size of 500, join density 0.25</td>
<td>64</td>
</tr>
<tr>
<td>5.2</td>
<td>Execution time with table size of 1000, join density 0.25</td>
<td>67</td>
</tr>
<tr>
<td>5.3</td>
<td>Execution time with table size of 2500, join density 0.25</td>
<td>69</td>
</tr>
<tr>
<td>5.4</td>
<td>Execution time with table size of 5000, join density 0.25</td>
<td>70</td>
</tr>
<tr>
<td>5.5</td>
<td>Execution time with table size of 10000, join density 0.25</td>
<td>72</td>
</tr>
<tr>
<td>5.6</td>
<td>Execution time with table size of 500, join density 0.5</td>
<td>74</td>
</tr>
<tr>
<td>5.7</td>
<td>Execution time with table size of 1000, join density 0.5</td>
<td>74</td>
</tr>
<tr>
<td>5.8</td>
<td>Execution time with table size of 2500, join density 0.5</td>
<td>74</td>
</tr>
<tr>
<td>5.9</td>
<td>Execution time with table size of 5000, join density 0.5</td>
<td>78</td>
</tr>
<tr>
<td>5.10</td>
<td>Execution time with table size of 10000, join density 0.5</td>
<td>79</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Opening Remarks

We are living in the middle of an electronic “Information Revolution”. The printed medium which has so far dominated the storage and dissemination of information is being supplanted by the electronic medium. Our store of knowledge itself has not grown that dramatically; what has in fact happened is that the numerous repositories of information have been linked to one another and been made accessible to anyone possessing the basic hardware and software.

The most common type of store in the electronic medium is a database system. The information in this type of store is well-managed and easy to get to in a systematic manner. The great advance that we have made in networking various data repositories has brought with it several thorny issues which need to be resolved if we are to take full advantage of all the information that has been suddenly made available to us.

A basic problem with our current network of information sources is that of incompatibility. Tools and methods of interacting with the data in one store need not work in another store. The pressing need is for “interoperable” systems which would be able to manage multiple information sources in a seamless manner and enable users to pose search queries which are applicable over the entire network of information sites.

Even within the same information source, within a well-structured environment such as a database management system (DBMS), a query against one database may be “incompatible” against another. This is because what is considered as “data”
in one database could be seen as part of the "schema" in another. Existing query languages do not allow the user to ask for elements of the schema itself to form part of the output.

It is in this context that we introduce SchemaLog, a powerful language for advanced database logic programming first proposed by Lakshmanan et al., in [LSS93, LSS96] and later expatiated upon by Subramanian in [Sub97]. It was established there that SchemaLog can offer a powerful platform in a variety of settings including multi-database interoperability, database programming with schema browsing, cooperative query answering, computing forms of aggregate queries which are beyond the scope of conventional database query languages, database restructuring, and, in the context of the World Wide Web (WWW), querying and restructuring Web pages. SchemaLog has a higher order syntax, but has a first-order semantics. Indeed, it has a sound and complete proof theory as shown in [LSS96], [Sub97].

A prototype platform for interoperability among a number of INGRES databases was completed recently [LSPS95], based on a fragment of SchemaLog.

The theoretical basis for the use of SchemaLog as a powerful database programming language even in the context of a single database was established in the papers we have already referred to earlier. The main goal of this thesis, is to establish the practical basis for this claim by describing efficient architectures and algorithms for the implementation of SchemaLog as a database programming language, in the context of a single database.

In the efficient implementation of a language like SchemaLog, we can distinguish two types of optimization issues:

- problems peculiar to the querying of a federation of homogeneous/heterogeneous databases.

- problems that arise even within the context of a single database because of requirements such as data/meta-data interaction and the dynamic restructuring of data and schema.

These two types of issues are clearly orthogonal and can be tackled independently of each other. The first area has been partially studied recently in [LSPS95] which provided a prototype platform for interoperability among a number of INGRES databases. The focus in this thesis is on the second area, that of implementation issues in a single database context.
1.2 Outline of the thesis

The major goal of this thesis has been to work towards an implementation of SchemaLog as a database programming language. Although SchemaLog as a language has the capability of addressing a federation of databases, we have restricted ourselves in this study, as mentioned earlier, to the context of a single database.

1. Since our implementation is based on the SchemaLog language, which is a recently-developed language and not that widely-known, we have gone to some detail in Chapter 2 to give the reader sufficient information on the syntax and semantics of the language to understand what follows in the rest of the thesis. Later in the chapter, we present the new operators in Schema Algebra (SA), with examples to illustrate their use. The basic theoretical notions in this chapter are derived from [LSS96], and [Sub97]. We have made several modifications to the syntax and semantics of the operators in order to suit a single database context. New and detailed examples have been provided to illustrate the working of the operators as well as the translation of a sample SchemaLog program into the procedural level of algebraic operators.

2. For database applications, it is important that querying and restructuring be implemented in a set-oriented manner, as opposed to the “tuple-at-a-time” paradigm of Prolog. In chapter 3, we present the theoretical basis for a top-down implementation of SchemaLog following the set-oriented Rule/Goal Tree (RGT) evaluation method of Datalog. Detailed algorithms for the RGT evaluation are set forth, and an example of a Rule-Goal Tree for a specific SchemaLog program has also been worked out.

3. Any implementation, if it is to be efficient, needs to have suitable storage structures. Since SchemaLog treats schema elements on the same level as data elements, traditional storage structures may prove unsuitable. With this in mind, we present, in chapter 4, three different physical storage architectures. For each of these architectures we have developed new algorithms to implement the various operators in SA. For each algorithm, a theoretical cost estimate has also been done.
4. To test the practical efficacy of the storage strategies introduced in chapter 4, we implemented a few chosen \textit{S.A} operators (according to the algorithms presented in Chapter 4) in all the three strategies using the MS-Access DBMS. We ran a number of experiments, varying such parameters as table size, join density and selection density. The results of these experiments were tabulated and also plotted on graphs. Details of these experiments form the contents of chapter 5.

5. Chapter 6 concludes our study with a summary, a comparison of our work to similar other practical implementations, and some pointers to future research in the area.

1.3 Contributions of this thesis

We have just seen an overview of the contents of this thesis. We conclude this introductory chapter with a listing of what we consider to be the significant contributions of this thesis.

1. \textit{SchemaLog} had been designed by its authors to suit a variety of needs in database programming in such diverse environments as multi-database interoperability and world-wide-web querying. In order to study in detail the issues peculiar to a single database setting, we have, in this thesis, modified the \textit{SchemaLog} syntax and semantics to suit such a context.

2. Schema Algebra as proposed in [Sub97] contains complexities of syntax which are not required in a single database setting. We have, therefore, in this thesis developed a simplified version of the Algebra needed for a procedural view of the high-level \textit{SchemaLog} language.

3. Being a database programming language, it is important that an evaluation of a \textit{SchemaLog} program yield a set of tuples in the output. To achieve this we have proposed a top-down evaluation based on the well-known Rule-Goal Tree method. We have devised the algorithms needed to implement this method in the case of \textit{SchemaLog} programs. We have also defined the role that the Schema Algebra operators play within the top-down evaluation process.
4. To improve the efficiency of the evaluation process, we have proposed three different ways of storing and accessing the data used in a SchemaLog program. One of them is an adaptation of the conventional storage strategy, and the other two have been specially devised to have relevance in the context of the implementation of the "restructuring" operators of SchemaLog.

5. In each of the three strategies mentioned, above we have evolved algorithms to implement some of the key operators in Schema Algebra. A theoretical estimate of the cost involved in an implementation of each of the algorithms is also worked out.

6. Finally, we ran a number of experiments to test the performances of the various storage strategies under varying conditions of data size and data contents. Based on our detailed tabulations of the results, we were able to provide valuable pointers as to the best implementation strategies to be followed.
Chapter 2

Basics of *SchemaLog* and Schema Algebra

2.1 Introduction

This chapter provides a condensed view of those aspects of *SchemaLog* which are needed for a proper understanding of this thesis. Section 2.2 gives an abbreviated view of the syntax of *SchemaLog*. The semantics of the language is illustrated by means of an example later in the section. Section 2.3 is devoted to a more detailed discussion of the Algebraic Operators in *Schema Algebra*, an algebra that has been developed in order to implement the *SchemaLog* language. The chapter concludes with a section containing a further examples to illustrate *SchemaLog* rules and Schema Algebra operators.

2.2 Syntax of *SchemaLog*

We introduce here the syntax of *SchemaLog* and illustrate the semantics informally via examples. The syntax we will use is an adaptation of the full *SchemaLog* given in [Sub97] and [LSS96] to a single database context. For a full and formal account, the reader is referred to [Sub97].

*SchemaLog* features three\(^1\) kinds of basic expressions, called *atoms*. Their notation as well as an informal meaning is given below.

\(^1\)Actually, full *SchemaLog* features four. In a single database context, these reduce to three.
• \( (\text{rel}) \) — there is a relation named \( (\text{rel}) \) in the database.

• \( (\text{rel})[(\text{attr})] \) — there is a relation named \( (\text{rel}) \) in the database, and the schema of \( (\text{rel}) \) includes an attribute named \( (\text{attr}) \).

• \( (\text{rel})[(\text{tid}) : (\text{attr}) \rightarrow (\text{val})] \) — there is a relation named \( (\text{rel}) \) in the database, and the schema of \( (\text{rel}) \) includes an attribute named \( (\text{attr}) \), and furthermore \( (\text{rel}) \) contains a tuple \( (\text{tid}) \) which has value \( (\text{val}) \) under column \( (\text{attr}) \).

In the above, \( (\text{rel}) \), \( (\text{tid}) \), \( (\text{attr}) \), and \( (\text{val}) \) are arbitrary first-order terms, built up as usual from a vocabulary including constants, function symbols, and variables. In this thesis, for simplicity, we shall deal only with function-free SchemaLog.

Molecules are expressions of the form \( (\text{rel})[(\text{tid}) : (\text{attr})_1 \rightarrow (\text{val})_1, \ldots, (\text{attr})_n \rightarrow (\text{val})_n] \) and are a syntactic sugar for the conjunction \( (\text{rel})[(\text{tid}) : (\text{attr})_1 \rightarrow (\text{val})_1] \land \cdots \land (\text{rel})[(\text{tid}) : (\text{attr})_n \rightarrow (\text{val})_n] \). The \( (\text{tid}) \) acts as a "glue" to combine the pieces from different atoms together. Existential "don't care" tid variables appearing in rule bodies can be omitted (see Example 2.1).

Besides these, SchemaLog also uses programming predicates of the form \( p(t_1, \ldots, t_n) \) where \( p \) is an \( n \)-ary predicate symbol and \( t_i \) are terms. While programming predicates can be simulated using SchemaLog molecules, they add to programming convenience. We use the generic term database predicates to refer to SchemaLog atoms and molecules. The main difference between database predicates and programming predicates is that the former are used as an "interface" to relations — either existing in a database (i.e. the so-called EDB relations) or created by database programs written in SchemaLog (i.e. the so-called IDB relations) — whose schematic information (i.e. names of relations and their attributes) is regarded just as important as data, and needs to be queried and manipulated. On the other hand, programming predicates are used as a convenient device for storing intermediate results in computations, or sometimes for storing results of queries, where no particular attention is paid to the schema under which such results are stored. The term database relations refers to relations corresponding to database predicates, while programming relations refer to relations corresponding to programming predicates. When no confusion arises, we refer to database relations, simply as relations. It should be pointed out that in classical deductive database query languages such as Datalog, and actual prototypes which are based on them, the only kind of predicates supported are programming predicates.
As a result, the schematic information of all relations is essentially ignored in such languages.

[LSS96] provides a number of examples to illustrate the semantics of SchemaLog programs, and the various terminologies and conventions introduced above. In this section, we provide one example. A few more examples will follow later on in the Chapter, in Section 2.4

Example 2.1 Consider a stock exchange database that stores quarterly information on the companies traded in the exchange. The database also contains overall information on how the various industries are performing in each quarter. The following is the schema of this database.

\[
\begin{align*}
\text{stocks}(\text{comp}, \text{ticker}, \text{industry}) \\
\text{ind\_review}(\text{industry}, \text{quarter}, \text{asp}, \text{aeps}) \\
\text{ibm}(\text{quarter}, \text{asp}, \text{aeps}) \\
\text{msft}(\text{quarter}, \text{asp}, \text{aeps}) \\
\text{xon}(\text{quarter}, \text{asp}, \text{aeps})
\end{align*}
\]

The stocks relation contains information on each company, its ticker name and its type of business. The ind\_review relation maintains information on the average share price (asp) and the average earnings per share (aeps) of each business category, on a quarterly basis. The other relations in the database are named after the companies traded in the exchange and carry information related to the quarterly performance of the relevant company.

The following is a SchemaLog program that restructures the above database such that the restructured database presents a summary of all the information present in the original database.

\[
\begin{align*}
A[T : \text{quarter} \rightarrow Q, \text{ticker} \rightarrow T, A' \rightarrow V', \text{comp\_perf} \rightarrow V] & \leftarrow \text{stocks}[\text{ticker} \rightarrow T, A' \rightarrow V'], A' \neq \text{ticker}, T[\text{quarter} \rightarrow Q, A \rightarrow V], A \neq \text{quarter} \\
A[X : \text{quarter} \rightarrow Q, \text{ind\_perf} \rightarrow V] & \leftarrow A[X : \text{industry} \rightarrow B], \\
\text{ind\_review}[\text{quarter} \rightarrow Q, \text{industry} \rightarrow B, A \rightarrow V], A \neq \text{quarter}, \\
A & \neq \text{industry}
\end{align*}
\]
The tables generated by the above rules are shown in Figure 2.1. Such a representation allows one to compare the quarterly performance of each company with the overall performance of the particular industry (e.g., computers) as a whole. A key point to note here is that the schema (and hence tuples) of the output tables is being assembled “piecemeal” by the above rules.

<table>
<thead>
<tr>
<th>quarter</th>
<th>ticker</th>
<th>company</th>
<th>industry</th>
<th>comp-perf</th>
<th>ind-perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>xon</td>
<td>Exxon Corp</td>
<td>oil</td>
<td>0.14</td>
<td>0.16</td>
</tr>
<tr>
<td>1</td>
<td>ibm</td>
<td>IBM Corp</td>
<td>computer</td>
<td>0.26</td>
<td>0.24</td>
</tr>
<tr>
<td>2</td>
<td>msft</td>
<td>Microsoft Corp</td>
<td>computer</td>
<td>0.28</td>
<td>0.25</td>
</tr>
</tbody>
</table>

... ... ... ... ... ... ...

<table>
<thead>
<tr>
<th>quarter</th>
<th>ticker</th>
<th>company</th>
<th>industry</th>
<th>comp-perf</th>
<th>ind-perf</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 2.1: Example Illustrating Database Restructuring

### 2.3 Operators in Schema Algebra

Our approach to the implementation of SchemaLog is based on compiling constructs in SchemaLog into corresponding operations in an extended version of relational algebra, called Schema Algebra, or SA for short. This algebra, described in detail in [Sub97] includes conventional relational algebra as a proper subset and features extensions that facilitate meta-data querying and restructuring. Specifically, SA consists of the following kinds of operations.

1. Classical RA operators: these are capable of querying all relations (programming as well as database).

2. Operators which take a programming relation as input parameter along with some additional parameters, query the data and schema of (database) relations
relative to the parameters and present the output as programming relations. They map relations in a database to programming relations.

3. Operators which take a programming relation as input, along with some parameters and restructure the information in the input relation according to the supplied parameters. Thus, these operators map programming relations into database relations.

Operators of type (2) and (3) are new and unique to $\mathcal{SA}$. In principle, $\mathcal{SA}$ can well be defined in the context of a federation of databases. Indeed, operations of type 1 and 2 were defined in such a context in [LSS96] and it was proved that they have an expressive power equivalent to that of SchemaLog programs containing only programming predicates in rule heads.

The main difference between classical RA operators and the new $\mathcal{SA}$ operators is that the latter treat both the schema of relations and the data in them in a uniform manner. In other words, the schema is given first class status. Classical RA has no means of retrieving or restructuring the schema of relations.

Throughout the thesis, we shall use the database pertaining to the New York Stock Exchange, shown in Figure 2.2, as a running example. The data is based on the actual data maintained at the URL http://www.ai.mit.edu/stocks.html. For brevity, only small parts of the actual database are shown.

### 2.3.1 Definitions of Operators

The classical relational algebra operators — selection, projection, join, union, intersection, difference, cartesian product – are defined in the usual manner.

We give below the definitions of the operators of type (2) and (3). The definitions below, although based on those given in [Sub97], have been modified to fit into a single database context. The algorithms and the implementations described in Chapters 4 and 5 also assume a single database environment.

We start with a descriptive definition of what a database is, in the SchemaLog context. The Herbrand Base $HB$ is the set of all facts that we can express in the language of SchemaLog, i.e., all literals of the form $r[t : a \rightarrow v]$ such that $r, t, a$ and $v$ are constants. Then, all literals of $HB$ in which the value of $r$ is the name of a relation in the "Extensional Database" (i.e., the physically stored database) form
<table>
<thead>
<tr>
<th>Name</th>
<th>Ticker</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exxon Corp</td>
<td>xon</td>
<td>oil</td>
</tr>
<tr>
<td>IBM Corp</td>
<td>ibm</td>
<td>computer</td>
</tr>
<tr>
<td>Microsoft Corp</td>
<td>msft</td>
<td>computer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>93.62</td>
<td>92.62</td>
<td>92.62</td>
<td>2696.8</td>
</tr>
<tr>
<td>95/06/19</td>
<td>94.62</td>
<td>93.25</td>
<td>94.62</td>
<td>2078.7</td>
</tr>
<tr>
<td>95/06/20</td>
<td>98.00</td>
<td>93.87</td>
<td>97.75</td>
<td>3568.9</td>
</tr>
<tr>
<td>95/06/21</td>
<td>98.62</td>
<td>97.12</td>
<td>97.12</td>
<td>3580.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>72.25</td>
<td>71.37</td>
<td>72.12</td>
<td>3420.6</td>
</tr>
<tr>
<td>95/06/19</td>
<td>72.25</td>
<td>71.50</td>
<td>71.50</td>
<td>826.1</td>
</tr>
<tr>
<td>95/06/20</td>
<td>71.37</td>
<td>70.00</td>
<td>70.12</td>
<td>1267.3</td>
</tr>
<tr>
<td>95/06/21</td>
<td>70.00</td>
<td>68.75</td>
<td>69.25</td>
<td>1792.2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Date</th>
<th>High</th>
<th>Low</th>
<th>Close</th>
<th>Volume</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>87.50</td>
<td>84.87</td>
<td>87.00</td>
<td>5767.5</td>
</tr>
<tr>
<td>95/06/19</td>
<td>89.87</td>
<td>86.87</td>
<td>89.62</td>
<td>4410.1</td>
</tr>
<tr>
<td>95/06/20</td>
<td>91.37</td>
<td>89.75</td>
<td>91.37</td>
<td>3541.9</td>
</tr>
<tr>
<td>95/06/21</td>
<td>92.37</td>
<td>90.00</td>
<td>90.50</td>
<td>3583.8</td>
</tr>
</tbody>
</table>

Figure 2.2: The NYSE Database
what can be called \textit{EHB} (or, the extensional part of the Herbrand Base). It is this set of ground atoms that we call the "database". Note that in a \textit{SchemaLog} atom, the schema information about a relation is also available since the position of the constants in the atom denote whether it is a relation name, an attribute name or a value (the usual "data"). For a formal definition, please refer to [Sub97]. The idea is to view a database as a function, say \(d\), which when given a symbol, maps it to a function, say \(s\), which when given a symbol, maps it to another function, say \(v\), which when given a \(tid\), maps it to a value.

\textbf{Definition 2.1 (Fetching relation names)} \(\rho() = \{r \mid r \text{ is the name of a relation in the database}\}\).

Operator, \(\rho\), is a 0-ary operator. It returns as output the \textit{names} of all relations in the database.

E.g., \(\rho()\) would return the set \(\{\text{stocks, ibm, msft, xon}\}\), w.r.t. the stock market database of Figure 2.2. Notice that the output of \(\rho()\) includes both base and derived (database) relations, in general.

\textbf{Definition 2.2 (Fetching relations and their schemas)} \textit{Let \(s\) be a programming relation of arity \(k\) and \(i \leq k\) a positive number. Then \(\alpha_i(s) = \{(r, a) \mid r \in \pi_i(s), r \text{ is the name of a relation in the database, and } r \text{ has an attribute with name } a\}\).}

This operator, \(\alpha\), takes a programming relation as input and a column number as parameter. It then interprets the values appearing in that column as possible relation names in the database, and retrieves the names of their attributes. Note that \(\pi_i(s)\) denotes the classical projection.

As an example, suppose \(s = \{(\text{msft, micsrosoft, 100}), (\text{hp, hewlett-packard, 200})\}\). Then \(\alpha_1(s) = \{(\text{msft, date}), (\text{msft, high}), (\text{msft, low}), (\text{msft, close}), (\text{msft, volume})\}\). Notice that no output corresponding to \(hp\) is produced since \(hp\) does not correspond to any relation name in the database.

Before presenting the definition of the next operator, we need the notion of a \textit{pattern}, introduced in [LSS96] in a different context and further extended in [Sub97]. A \textbf{pattern} is of one of the following forms: \(\text{'}a\rightarrow v\text{'}\), \(\text{'}a \rightarrow \text{'}\), \(\text{'}\rightarrow v\text{'}\), or \(\text{'}\rightarrow \text{'}\). Intuitively, a pattern may be viewed as an attribute value pair, where either the attribute or the
value component (or both) of the pair could be missing. The next two operators defined here use a pattern to query data and meta-data in a database. This is achieved using a notion of satisfaction.

Let \( r \) be a relation, \( t \) a tuple in \( r \), and \( p \) a pattern. Then we say \( t \) satisfies \( p \) provided one of the following conditions holds:

- \( p \) is of the form \( a \rightarrow v \), and \( r \) has \( a \) as one of its attributes and \( t[a] = v \). In this case, the attribute value pair \((a, v)\) is said to be a witness pair.

- \( p \) is of the form \( a \rightarrow \ast \), and \( r \) has \( a \) as one of its attributes. In this case, the attribute value pair \((a, v)\) where \( t[a] = v \), is said to be a witness pair.

- \( p \) is of the form \( \rightarrow v \), and there is some attribute \( b \) in the schema of \( r \), such that \( t[b] = v \). In this case, for every attribute \( b \) for which \( t[b] = v \), \((b, v)\) is said to be a witness pair.

- \( p \) is of the form \( \ast \rightarrow \ast \). In this case, for every attribute \( a \) of \( r \), and every value \( v \) such that \( t[a] = v \), \((a, v)\) is a witness pair.

**Definition 2.3 (Querying data and meta-data)** Let \( s \) be any programming relation of arity \( k \), \( i \leq k \) any number, and let \( p \) be any pattern. Then \( \gamma_{i; \ast}(s) = \{(r, t, a, v) \mid r \in \pi_i(s), t \) is the id of a tuple that satisfies \( p \), and \((a, v)\) is an associated witness pair\}.

This operator, \( \gamma \), allows us to relate data to meta-data. It takes as input a programming relation, a column number and a pattern as parameters, and returns as output the details of all tuples in the database which satisfy the pattern.

E.g., let \( s = \{xon, ibm\} \). Then \( \gamma_{1; high \rightarrow}(s) = \{(xon, t4, high, 72.25), \ldots, (xon, t7, high, 70.00), (ibm, t8, high, 93.62), \ldots, (ibm, t11, high, 98.62)\} \).

Next we introduce an operator, which can be derived from the previous operators.

The main motivation for this operator is query processing efficiency. The analogy is that in classical relational algebra, the join is a derived operator which is helpful in efficient query processing, compared with the Cartesian product. This operator, \( \gamma^\wedge \), behaves essentially the same as \( \gamma \) except that it does not explicitly extract tuple ids, and it deals with a conjunction of patterns in one shot. We denote a conjunction of patterns as \( (p_1, \ldots, p_n) \), where each \( p_i \) is a pattern. Satisfaction of conjunctions of
patterns is defined in the obvious manner: a tuple satisfies a conjunctive pattern if it satisfies all patterns in the conjunction.

**Definition 2.4 (Querying conjunctive patterns)** Let \( s \) be a programming relation, \( i \) a column number, and \( (p_1, \ldots, p_n) \) a conjunctive pattern, where each \( p_j \) is of one of the forms — 'a→v', 'a→', '→v', or '→'. Then \( \gamma^\wedge_{i,(p_1,\ldots,p_n)}(s) = \{(r,a_1,v_1,\ldots,a_n,v_n) \mid r \in \pi_i(s), \text{ } r \text{ is the name of a relation in the database, } a_j \text{ 's are attributes in the schema of } r, \text{ and there is a tuple } t \in r \text{ such that } t[a_1,\ldots,a_n] = v_1,\ldots,v_n, \text{ and } t \text{ satisfies } (p_1,\ldots,p_n)\} \).

This operator takes a programming relation of arity \( k \) as input, a column number \( i \leq k \) and a conjunctive pattern as parameters, and returns as output the details of all parts of the database queried about using the conjunctive pattern.

E.g., let \( s = \{\text{stock}\} \). Then for the stock market database, \( \gamma^\wedge_{i,(\text{ticker} \rightarrow \rightarrow \text{computer})}(s) = \{\text{stock, ticker, ibm, type, computer}, (\text{stock, ticker, msft, type, computer})\} \).

The relation between \( \gamma \) and \( \gamma^\wedge \) can be expressed formally as:
\[
\gamma^\wedge_{i,(p_1,\ldots,p_n)}(s) = \sigma_{s2=s6\wedge\ldots\wedge s4(n-1)-2=s4n-2} (\gamma_{i,p_1}(s) \times \ldots \times \gamma_{i,p_n}(s))
\]

This completes the definition of operators that extract the information in relations and convert it into programming relations.

We next turn to the type (3) (restricting) operators.

**Definition 2.5 (Creating relations)** For a programming relation \( s \) and a column number \( i \), \( \kappa_i(s) \) creates a relation named \( r \) for each \( r \in \pi_i(s) \), if such a relation does not already exist.

In other words, the operator, \( \kappa \), takes a programming relation of arity \( k \) as input and a column number \( i \leq k \) as a parameter and creates relations with names corresponding to the entries appearing in column \( i \) of the input relation.

E.g., \( \kappa_1(\{(\text{close}), (\text{high}), (\text{low}), (\text{volume})\}) \) creates the relations close, high, low, volume (whose schema is not yet defined).

**Definition 2.6 (Creating relations with schemas)** For a programming relation \( s \) and column numbers \( i, j \), \( \varsigma_{i,j}(s) \) creates a relation \( r \) with attributes \( a_1, \ldots, a_n \) exactly when \( \sigma_{s1=s}(\pi_{i,j}(s)) = \{(r,a_1),\ldots,(r,a_n)\} \), whenever such a relation does not already exist.
This operator, $\zeta$, takes a programming relation of arity $k$ as input and two column numbers $i, j \leq k$ as parameters. It creates relations with names corresponding to the entries in column $i$ whose schemas are determined by interpreting the entries appearing column $j$ as the attributes associated with the relation names in column $i$.

For example, let $s = \{(\text{close}, \text{date}), (\text{close}, \text{ibm}), (\text{close}, \text{msft}), (\text{close}, \text{xon}), (\text{volume}, \text{date}), (\text{volume}, \text{ibm}), (\text{volume}, \text{msft}), (\text{volume}, \text{xon})\}$. Then $\zeta_{1,2}(r)$ will create two relations $\text{close}$ and $\text{volume}$ both with the schema $\{\text{date}, \text{ibm}, \text{msft}, \text{xon}\}$, and with no data, as shown in Figure 2.3.

![Figure 2.3: Example for creating relation schemas](image)

**Definition 2.7 (Creating and populating relations with schemas)** The operator $\varphi_{i,j,k,g_1,\ldots,g_m}(s)$ creates a relation $r$ with attributes $a_1, \ldots, a_n$ exactly when $\sigma_{s_1=r'}(\pi_{i,j}(s)) = \{(r, a_1), \ldots, (r, a_n)\}$, whenever such a relation does not already exist. Furthermore, it populates the relation $r$ with a tuple $t$ such that $t[a_1, \ldots, a_n] = (v_1, \ldots, v_n)$ exactly when $\exists t_1, \ldots, t_n \in r$ such that $t_i[i, j, k] = (s, a_i, v_i)$, $\pi_{i,j,k}(\{t_1, \ldots, t_n\}) = \{(s, a_1, v_1), \ldots, (s, a_n, v_n)\}$, and finally, $t_1[g_1, \ldots, g_m] = \cdots = t_n[g_1, \ldots, g_m]$. When the relation $s$ already exists, the tuples generated above are appended to this relation.

In less formal language, $\varphi$ takes a programming relation of arity $k$ as input, three column numbers $i, j, k$ and another list of column numbers $g_1, \ldots, g_m$ as parameters and returns as output several relations structured according to the interpretation of column $i$ entries as relation names, column $j$ entries as attribute names, and column $k$ entries as values. The facts thus produced form pieces of larger tuples which are put together based on equality on the columns $g_1, \ldots, g_m$.  

---

15
For example, let \( r = \{(close, \text{ibm}, 95/06/21, 97.12), \)
\( (close, \text{msft}, 95/06/21, 90.50), \)
\( (close, \text{xon}, 95/06/21, 69.25), \)
\( (close, \text{ibm}, 95/06/20, 97.75), \)
\( (close, \text{msft}, 95/06/20, 91.37), \)
\( (close, \text{xon}, 95/06/20, 69.25), \)
\( (close, \text{date}, 95/06/21, 95/06/21), \)
\( (close, \text{date}, 95/06/20, 95/06/20)\}.$

Then \( g_{1,2,4,3}(r) \) will create the relation shown in Figure 2.4.

<table>
<thead>
<tr>
<th>date</th>
<th>ibm</th>
<th>msft</th>
<th>xon</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/20</td>
<td>97.75</td>
<td>91.37</td>
<td>69.25</td>
</tr>
<tr>
<td>95/06/21</td>
<td>97.12</td>
<td>90.50</td>
<td>69.25</td>
</tr>
</tbody>
</table>

Figure 2.4: Example for creating and populating relations

It must be noted that this restructuring operator also has some limited update capabilities. Whenever the relation corresponding to a newly generated tuple already exists, the tuple is appended to such a relation (as a typical tuple insert).

For example, suppose that the relation close shown in Figure 2.4 already exists. Let \( s = \)
\( (close, \text{msft}, 95/06/19, 90.50), \)
\( (close, \text{xon}, 95/06/19, 70.25), \)
\( (close, \text{ibm}, 95/06/19, 99.75), \)
\( (close, \text{date}, 95/06/19, 95/06/19)\}.$

Then \( g_{1,2,4,3}(s) \) will have the effect of appending the tuple (95/06/19, 99.75, 90.50, 70.25) to the existing relation close.
2.4 Further Examples

We conclude this chapter with further examples which illustrate some of the features of SchemaLog programs and also show how the various operators defined above are actually used in the processing of a SchemaLog program to achieve desired results.

Example 2.2 The first example shows how queries involving data and meta-data can be expressed in SchemaLog.

Consider the query "Determine companies such that the low of Microsoft stock on some day is more than the high of the company's stock on the same day." In SchemaLog, this would be expressed as

\[ p(C_1, C_2) \leftarrow \text{stocks[name} \rightarrow C_1, \text{ticker} \rightarrow S_1], \text{stocks[name} \rightarrow C_2, \text{ticker} \rightarrow S_2], \right. \\
S_1[\text{date} \rightarrow D, \text{low} \rightarrow L], S_2[\text{date} \rightarrow D, \text{high} \rightarrow H], L > H. \]

\[ ?- p(\text{microsoft}, C_2). \]

Example 2.3 This example is a detailed demonstration of the working of the restructuring operator, $\rho$, with regard to a given input.

Consider the stock market database of Figure 2.2. The database stores information related to individual stocks in different (database) relations, organised by the stock ticker symbol. Figure 2.4 shows a restructured view of the information in the original database. It permits a quick comparison of the closing prices of different stocks on a given day. Such a restructuring is often referred to as cross-tabulation (crosstabs) [GBLP96]. This restructuring can be done readily by writing the one-line SchemaLog program

\[ \text{close[D : date} \rightarrow D, S \rightarrow P] \leftarrow S[\text{date} \rightarrow D, \text{close} \rightarrow P], \text{stocks[ticker} \rightarrow S]. \]

In this program, both retrieving information from the input EDB, and presenting the output are done through SchemaLog molecules. The rule computes attribute value pairs forming any one tuple in the output relation close in a piecemeal manner. The tuple id $D$ acts as a "glue" and makes sure that all pieces related to one tuple in the output relation close are correctly grouped together into one tuple.

For example, for the stock market data of Figure 2.2, the facts computed for close are:

\[ \text{close[95/06/21 : date} \rightarrow 95/06/21, \text{ibm} \rightarrow 97.125], \]
\[ \text{close[95/06/21 : date} \rightarrow 95/06/21, \text{msfi} \rightarrow 90.500], \] and
close[95/06/21 : date→95/06/21, xon→69.250].

Since all three facts have the same tuple id 95/06/21, they are correctly grouped into one tuple

\[
\text{close}[95/06/21 : \text{date}\rightarrow 95/06/21, \text{ibm}\rightarrow 97.125, \text{msft}\rightarrow 90.500, \text{xon}\rightarrow 69.250].
\]

**Example 2.4** This example takes a generalized version of the program in Example 2.3 and illustrates how a SchemaLog program can be rewritten in terms of the \(SA\) operators we have described in this chapter.

Suppose it is required to restructure the information in the stock market database such that it is readily possible to compare the relative performance of the various stocks with respect to each of their properties – low, high, and closing prices and volume of trading. The intended effect is that there would be separate relations organized around each of the above properties which list the performances of various stocks in each row, on a per day basis. It should be pointed out that such a query/restructuring is not a whim. Indeed in stock market applications, such restructuring arises naturally. In fact, the traditional approach to restructuring information is to write down special code for this purpose. The disadvantage of this approach is that such code tends to be specialized, \textit{ad hoc}, and rather complex. Thus it does not permit general forms of restructuring and does not admit easy modification. (See the URL http://www.ai.mit.edu/stocks.html for details.)

In order to create separate relations corresponding to each of the properties close, high, low and volume we need to make only a minor modification to the SchemaLog program of Example 2.3.

\[
A[D : \text{date}\rightarrow D, \ S\rightarrow P] \leftarrow S[\text{date}\rightarrow D, \ A\rightarrow P], \ A \neq \text{date}, \ \text{stocks}[\text{ticker}\rightarrow S].
\]

This program can be rewritten in the form of the following algebraic expression, using the \(SA\) operators defined earlier:

\[
\varnothing_{4,1,5,3}((\sigma_{\text{\$4}\neq \text{\'date'}} (\gamma_{1;\text{date}\rightarrow,\rightarrow} (\pi_{\text{ticker}}(\text{stocks})))) \cup (\pi_{2,2,3,4,3}(\gamma_{1;\text{date}\rightarrow,\rightarrow} (\pi_{\text{ticker}}(\text{stocks})))))
\]

The computation of the rather complex algebraic expression above can be easily understood if broken down into a sequence of simpler computations storing intermediate results. In the steps given below, \(r_1\) to \(r_5\) are temporary relations.

18
1. $r_1 := \pi_{ticker}(stocks)$
2. $r_2 := \gamma^\wedge_{\langle date \rightarrow, 
\rightarrow \rangle}(r_1)$;
3. $r_3 := \sigma_{\langle date \rangle}(r_2)$;
4. $r_4 := \pi_{2,2,3,4,3}(r_2)$;
5. $r_5 := r_3 \cup r_4$;
6. $\varrho_{4,1,5,3}(r_5)$;

A detailed study of the result of each step above will help us to understand the working of the $\mathcal{SA}$ operators, especially $\gamma^\wedge$ and $\varrho$.

The result of Step 1 (above) is the relation $r_1$ which is given in Figure 2.5.

<table>
<thead>
<tr>
<th>stocks</th>
</tr>
</thead>
<tbody>
<tr>
<td>ticker</td>
</tr>
<tr>
<td>xon</td>
</tr>
<tr>
<td>ibm</td>
</tr>
<tr>
<td>msft</td>
</tr>
</tbody>
</table>

Figure 2.5: $r_1$: the result of Step 1

In Step 2, every relation mentioned in $r_1$ of figure 2.5 is taken up and every row of these relations is processed to fit the given pattern. For the sake of brevity, we have processed only one row from each of the relations. The result is $r_2$ as given in Figure 2.6.

Step 3 removes those rows from $r_2$ where the fourth column contains the string “date”. The result is $r_3$ as in Figure 2.7.

Step 4 eliminates some of the columns in $r_2$ and re-orders the other columns, and produces the relation $r_4$ in Figure 2.8.

Step 5 does a regular union of the tables $r_3$ and $r_4$ and writes the result in $r_5$ as in Figure 2.9.

Finally, in Step 6 the $\varrho$ operator is applied to $r_5$ according the interpretation of the input parameters. The result is the set of tables given in Figure 2.10. Please note that processing only the rows of $r_5$ will produce merely the first row of each of the tables in figure 2.10. This is because in Step 2 we did not process all the rows in the
<table>
<thead>
<tr>
<th>xon</th>
<th>date</th>
<th>95/06/16</th>
<th>date</th>
<th>95/06/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>high</td>
<td>72.25</td>
</tr>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>71.37</td>
</tr>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>72.12</td>
</tr>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>3420.6</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>date</td>
<td>95/06/16</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>high</td>
<td>93.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>92.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>92.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>2696.8</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>date</td>
<td>95/06/16</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>high</td>
<td>87.50</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>84.87</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>87.00</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>5767.5</td>
</tr>
</tbody>
</table>

Figure 2.6: $r_2$: the result of Step 2

<table>
<thead>
<tr>
<th>xon</th>
<th>date</th>
<th>95/06/16</th>
<th>high</th>
<th>72.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>71.37</td>
</tr>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>72.12</td>
</tr>
<tr>
<td>xon</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>3420.6</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>high</td>
<td>93.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>92.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>92.62</td>
</tr>
<tr>
<td>ibm</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>2696.8</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>high</td>
<td>87.50</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>84.87</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>87.00</td>
</tr>
<tr>
<td>msft</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>5767.5</td>
</tr>
</tbody>
</table>

Figure 2.7: $r_3$: the result of Step 3

<table>
<thead>
<tr>
<th>date</th>
<th>date</th>
<th>95/06/16</th>
<th>high</th>
<th>96/06/16</th>
</tr>
</thead>
<tbody>
<tr>
<td>date</td>
<td>date</td>
<td>95/06/16</td>
<td>low</td>
<td>96/06/16</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
<td>95/06/16</td>
<td>close</td>
<td>96/06/16</td>
</tr>
<tr>
<td>date</td>
<td>date</td>
<td>95/06/16</td>
<td>volume</td>
<td>96/06/16</td>
</tr>
</tbody>
</table>

Figure 2.8: $r_4$: the result of Step 4

20
input. The other rows in figure 2.10 are the result of the complete processing of the input database of Figure 2.2.

A comparison of the database of Figure 2.2 with the one obtained in Figure 2.10 is a good illustration of the power and usefulness of the $\phi$ operator. It can be seen that the columns of figure 2.2 have become tables in figure 2.10 and the tables of figure 2.2 have become columns in figure 2.10, with the data suitably reallocated.

### 2.5 Concluding Remarks

This chapter provided a basic introduction to the high-level database programming language called *SchemaLog*. The syntax of the language was summarised and its semantics illustrated by some examples (in Sections 2.2 and 2.4). Section 2.3 was a rather detailed treatment of the algebraic operators needed to translate a *SchemaLog* program into an equivalent algebraic expression. Both traditional Relational Algebraic Operators as well as some new ones specific to the needs of *SchemaLog* were defined and their semantics illustrated with examples where necessary. Finally in
<table>
<thead>
<tr>
<th>Date</th>
<th>xon</th>
<th>ibm</th>
<th>msft</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>72.25</td>
<td>93.62</td>
<td>87.50</td>
</tr>
<tr>
<td>95/06/19</td>
<td>72.25</td>
<td>94.62</td>
<td>89.87</td>
</tr>
<tr>
<td>95/06/20</td>
<td>71.37</td>
<td>98.00</td>
<td>91.37</td>
</tr>
<tr>
<td>95/06/21</td>
<td>70.00</td>
<td>98.62</td>
<td>92.37</td>
</tr>
</tbody>
</table>

**high**

<table>
<thead>
<tr>
<th>Date</th>
<th>xon</th>
<th>ibm</th>
<th>msft</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>71.37</td>
<td>92.62</td>
<td>84.87</td>
</tr>
<tr>
<td>95/06/19</td>
<td>71.50</td>
<td>93.25</td>
<td>86.87</td>
</tr>
<tr>
<td>95/06/20</td>
<td>70.00</td>
<td>93.87</td>
<td>89.75</td>
</tr>
<tr>
<td>95/06/21</td>
<td>68.75</td>
<td>97.12</td>
<td>90.00</td>
</tr>
</tbody>
</table>

**low**

<table>
<thead>
<tr>
<th>Date</th>
<th>xon</th>
<th>ibm</th>
<th>msft</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>72.12</td>
<td>92.62</td>
<td>87.00</td>
</tr>
<tr>
<td>95/06/19</td>
<td>71.50</td>
<td>94.62</td>
<td>89.62</td>
</tr>
<tr>
<td>95/06/20</td>
<td>70.12</td>
<td>97.75</td>
<td>91.37</td>
</tr>
<tr>
<td>95/06/21</td>
<td>69.25</td>
<td>97.12</td>
<td>90.50</td>
</tr>
</tbody>
</table>

**close**

<table>
<thead>
<tr>
<th>Date</th>
<th>xon</th>
<th>ibm</th>
<th>msft</th>
</tr>
</thead>
<tbody>
<tr>
<td>95/06/16</td>
<td>3420.6</td>
<td>2698.8</td>
<td>5767.5</td>
</tr>
<tr>
<td>95/06/19</td>
<td>826.1</td>
<td>2078.7</td>
<td>4410.1</td>
</tr>
<tr>
<td>95/06/20</td>
<td>1267.3</td>
<td>3568.9</td>
<td>3541.9</td>
</tr>
<tr>
<td>95/06/21</td>
<td>1792.2</td>
<td>3580.5</td>
<td>3583.8</td>
</tr>
</tbody>
</table>

**volume**

Figure 2.10: The result of Step 6
Section 2.4, Example 2.4 we showed how a SchemaLog program can be expressed in an equivalent algebraic expression. We concluded with a detailed example of the step-by-step evaluation of a SchemaLog rule.

In a subsequent chapter, we shall provide algorithms for implementing the $SA$ operators in different storage strategies. A program and its equivalent algebraic expression, such as the one in Example 2.4, may then be readily implemented in the required storage format.
Chapter 3

Top-Down Processing of SchemaLog Programs

3.1 Introduction

In this chapter we discuss the top-down processing of SchemaLog programs. In particular, we investigate how the set oriented Rule/Goal Tree (RGT) evaluation method [Ull89] proposed for classical logic can be extended to the SchemaLog setting. Our choice of this methodology is due to the fact that set-oriented query processing techniques are more suitable for database applications as opposed to the tuple-at-a-time paradigm of Prolog. At the guts of the algorithm we discuss here, lie the $SA$ operators defined in the previous chapter.

The rule/goal tree(RGT) is a representation of the sequence of exploration of the goals, their rules, and their subgoals required to answer a query. The RGT evaluation method is based on a depth-first search of the rule/goal tree in which a set of tuples of bindings is obtained at each node of the tree. Details of the rule/goal tree as well as the attendant query processing algorithm for classical logic can be found in [Ull89].

The notion of unification plays an important role in the construction of RGTs. Unification in SchemaLog is different from its classical counterpart mainly because SchemaLog requires unification of "literals" of unequal depth. [LSS96] discusses this issue at depth and presents an algorithm for computing the most general unifier (mgu)
of two SchemaLog atoms. Based on the SchemaLog notion of unification, the conventional algorithm for constructing the RGT of a program can be adapted to serve our purpose of evaluating SchemaLog programs. An important consequence of the fact that SchemaLog unification is performed on atoms is that we need to 'atomize' a SchemaLog program (which in general might contain molecules) before applying the top-down algorithm. Example 3.2 illustrates this point. We give an algorithm to "atomize" any given SchemaLog program.

The major strength of SchemaLog lies in its ability to express novel querying as well as powerful restructuring operations. Our adaptation of the classical RGT evaluation algorithm accounts for these unique features of SchemaLog. In the following, we sketch the major issues that arise in the development of such an algorithm. After that, we present a comprehensive algorithm for the RGT evaluation of a SchemaLog program. We conclude with an example of a RGT expansion of a SchemaLog program.

### 3.2 Atomizing a SchemaLog program

The use of "molecules" in a SchemaLog program makes for user convenience in reading and writing programs. But when the program is to be evaluated in a top-down paradigm, we need to use the program as input to such algorithms as unification (discussed in [Sub97]) and Rule-Goal Tree Processing (discussed later in this Chapter). Since these algorithms accept only SchemaLog atoms as input, all molecules in a given SchemaLog program need to be atomized as a necessary pre-processing step to top-down evaluation. In this section we first present an algorithm for "atomization" and then illustrate its use with an example.

**Algorithm 3.1 Atomization of a SchemaLog program**

**Input:** A SchemaLog program

**Output:** A SchemaLog program that contains no "molecules"

begin

for every rule in the given program

if the body of the rule contains a molecule
rewrite the body as a new body, $B$, such that

every molecule of the form $r[t : a_1 \rightarrow v_1, ..., a_n \rightarrow v_n]$
is written as $r[t : a_1 \rightarrow v_1] \land \ldots, \land r[t : a_n \rightarrow v_n]$

if the molecule does not have a tuple-id,
supply a tuple-id, distinct from the ones used for other molecules.

if the head of the rule contains a molecule of the form

$r[t : a_1 \rightarrow v_1, ..., a_m \rightarrow v_m]$

construct $m$ rules of the form:

$r[t : a_1 \rightarrow v_1] \leftarrow B$

\[\ldots \quad \ldots\]

$r[t : a_m \rightarrow v_m] \leftarrow B$

end

We illustrate the working of the above algorithm with the following example:

**Example 3.1** Let us revisit the one-rule *SchemaLog* program of Example 2.3:

$$
\text{close}[D : date \rightarrow D, S \rightarrow P] \leftarrow S[\text{date} \rightarrow D, \text{close} \rightarrow P], \text{stocks}[\text{ticker} \rightarrow S].
$$

This rule contains molecules both in the head and in the body. Applying the *Atomization Algorithm* to this rule yields the following two rules:

$$
\text{close}[D : date \rightarrow D] \leftarrow S[T : date \rightarrow D], S[T : close \rightarrow P], \text{stocks}[\text{ticker} \rightarrow S]
$$

$$
\text{close}[D : S \rightarrow P] \leftarrow S[T : date \rightarrow D], S[T : close \rightarrow P], \text{stocks}[\text{ticker} \rightarrow S]
$$

As we can see, the two rules above do not contain any molecules.

### 3.3 Outline of the Top-Down Processing procedure

At the very outset we should note that the area of top-down processing of logic programs is a well-studied one. The relevant algorithms have been discussed and analyzed in detail in logic programming literature. We shall, therefore, model our
algorithms on the classical ones and make modifications and additions to them for the specific requirements of SchemaLog.

There are two operations which are often invoked during the top-down evaluation of SchemaLog programs — one for converting the database relations corresponding to (programming) subgoals to relations over variables mentioned in that subgoal, and the other for converting a (programming) relation for the body to a relation for the head by translating from the viewpoint of variables to the viewpoint of arguments. For Datalog (which is the classical programming language for deductive databases)\(^1\), this switching between argument and variable viewpoints is accomplished by means of procedures called atov() and vtoa() [Ull89]. As SchemaLog atoms are syntactically different from their classical counterparts, the atov() and vtoa() procedures are somewhat different for our setting. Our approach efficiently realizes these same operations by using the technique of first reducing the database predicate arguments to a template that corresponds to a conventional predicate, and then applying the classical version of the operations.

For instance, in our a2v() procedure, in order to convert a relation \(M\) into a relation whose attributes correspond to variables appearing in a SchemaLog atom \(A\) of the form \(\alpha_1[\alpha_2 : \alpha_3 \rightarrow \alpha_4]\), \(A\) is reduced to a template \(\text{temp}(\alpha_1, \alpha_2, \alpha_3, \alpha_4)\) and the conventional atov() algorithm with the template and \(M\) as arguments is applied. The adaptation of the vtoa() algorithm is similar in nature.

At the heart of the RGT evaluation algorithm lies two mutually recursive procedures — expand_goal() and expand_rule(). Given a SchemaLog goal \(G\) and a relation \(M\) that provides bindings for variables in \(G\), expand_goal() returns a relation \(R\) that is the set of tuples (bound by \(M\)) that can be inferred from the database using the program, and match \(G\). expand_rule() on the other hand, takes a rule \(r\) and the initial bindings for the variables in this rule and generates a relation \(R\) that is the set of tuples inferred from the database and the rule. Further, this procedure performs the restructuring dictated by \(R\) and the head predicate of rule \(r\). Thus the querying and restructuring facets of SchemaLog query processing are neatly decoupled in procedures expand_goal() and expand_rule() respectively.

   expand_goal() invokes the querying operations in \(SA\) via a procedure called

\(^1\)For an introduction to Datalog, refer to [CGT89]
query_goal(). Corresponding to each type of SchemaLog atom, this procedure invokes an appropriate $\mathcal{SA}$ expression involving type (2) operations. For instance, the call query_goal($X[T : a \rightarrow V]$) invokes the operation $\sigma_{\gamma_1 \gamma_2 \gamma_3 \cdots \gamma_r} \gamma_1 \gamma_2 \gamma_3 \cdots \gamma_r \rightarrow (p)$. Procedure restructure_head(), called from expand_rule(), invokes an $\mathcal{SA}$ expression that includes a type (3) operation corresponding to the head predicate of the SchemaLog rule under expansion.

As in the classical case, a queue-based version of this algorithm based on a breadth-first search of the RGT can be realized by queueing the calls to expand_goal() and expand_rule() rather than stacking them.

### 3.4 Top-Down Algorithm for SchemaLog

In this section, we present the top-down query processing algorithm for SchemaLog.

**Algorithm 3.2 rule/goal tree evaluation**

**Input:** A SchemaLog program, a database $D$, a query $G_0$, and the relation $M_0$ that provides bindings for zero or more arguments of $G_0$.

**Output:** A set of tuples of bindings for variables in $G_0$, that satisfy the query against $D$.

**Body:** The main components of the algorithm are procedures query_goal(), restructure_head(), and two mutually recursive procedures expand_goal() and expand_rule(). To account for the higher-order syntax of SchemaLog, procedures $\lambda 2v()$ and $v2\lambda()$ are implemented differently from their classical counterpart.

---

**procedure $\lambda 2v(A, M)$**

**Input:** A SchemaLog atom $A$ of the form $\beta_1[\beta_2 : \beta_3 \rightarrow \beta_4]$ or an atom of a lesser depth, and a relation $M$.

**Output:** A relation whose attributes are variables in $A$.

**begin**

- construct a template temp($\beta_i, \ldots, \beta_k$), $1 \leq i \leq k \leq 4$, $\beta_j$ is a variable corresponding to an attribute of $M$ and appears in $A$;
- let $X_1, \ldots, X_n$ be the distinct variables among $\beta_i, \ldots, \beta_k$;
- let $Q$, the output relation have scheme $X_1, \ldots, X_n$ and be empty;
for each tuple \( t \) in \( M \)
begin
if there is a term matching \( \tau \) for \( \text{temp}(\beta_i, \ldots, \beta_k) \) and tuple \( t \),
add to \( Q \), the tuple \((\tau(X_1), \ldots, \tau(X_n))\)
end
return \( Q \)
end

procedure \( v2a(A, R) \)
Input: A SchemaLog atom \( A \) of the form \( \beta_1[\beta_2 : \beta_3 \rightarrow \beta_4] \) or an atom of a lesser depth, and a relation \( R \) with scheme \((X_1, \ldots, X_n)\).
begin
construct a predicate \( \text{temp}(\beta_i, \ldots, \beta_k) \), \( 1 \leq i \leq k \leq 4 \), where \( \beta_i, \ldots, \beta_k \) are all the variable occurrences in \( A \) and \( \beta_j \) is an attribute of \( R \);
let \( S \), the output relation be empty;
for each tuple \( t \) of relation \( R \)
begin
for each variable \( X \) appearing in \( \text{temp} \), replace all occurrences of \( X \) in \( \text{temp} \) by \( t[X] \);
add the resulting tuple \((t_1, \ldots, t_m)\) to \( S \)
end
return \( S \)
end

procedure \( \text{expand-goal}(M, G, R) \)
begin
if \( M = \phi \) then
begin
\( R = \phi \); return;
end
end
\( R = \phi \);
for each rule \( r \) with head \( H \) such that \( G \) is unifiable to \( H \)

\[29\]
begin
let \( \tau \) be the mgu from \( G \) to \( H \);
compute \( S_0 = \lambda Z (\tau(H), M) \);
\text{EXPAND\_RULE}(S_0, \tau(\tau));
end
\( R = \text{QUERY\_GOAL}(G) \cong M \);
end

---

procedure \text{EXPAND\_RULE}(S_0, r)
begin
let \( r = H \leftarrow G_1, \ldots, G_k \);
for \( i = 1 \) to \( k \) do
begin
\( M_i = \lambda Z (G_i, S_{i-1}) \);
\text{EXPAND\_GOAL}(M_i, G_i, R_i);
\( Q_i = \lambda Z (G_i, R_i) \);
\( S_i = \prod_T (S_{i-1} \cong Q_i); /* T is the set of variables that appear in the scheme of \( S_{i-1} \) or \( Q_i \), and also appear in one of \( H, G_{i+1}, \ldots, G_k */ \)
end
\text{RESTRUCTURE\_HEAD}(H, S_k)
end

---

procedure \text{QUERY\_GOAL}(A)
Input: A SchemaLog atom of the form \( \beta_1[\beta_2 : \beta_3 \rightarrow \beta_4] \), or an atom of a lesser depth.
begin
\text{case} \( A \) \text{ is of depth:}
1: \text{if} \( \beta_1 \) \text{ is a constant, return } \sigma_{S_1=\beta_1}(\rho)
else \text{return } \rho;
2: \text{return } \sigma_{A, S_1=\beta_i, \alpha S_1}(\rho), \ \beta_i \text{ is a constant;}
3: \text{return } \sigma_{A, S_1=\beta_i, \gamma S_1} \rightarrow (\rho), \ \beta_i \text{ is a constant}
end
procedure restructure_head(A, P)
Input: A SchemaLog atom of the form $\beta_1[\beta_2 : \beta_3 \rightarrow \beta_4]$, or an atom of a lesser depth, and a relation P whose attributes correspond to variables in A.

Note: The unary function $\nu$ used below, takes as argument a variable appearing in A and returns the position of its corresponding attribute in P.

begin
  case A is of depth
  1: if $\beta_1$ is a constant, return $\kappa_1\{\beta_1\}$
     else return $\kappa_{\nu(\beta_1)}(P)$;
  2: form a tuple $t = <\beta_i, \ldots, \beta_k>$, $1 \leq i \leq k \leq 2$, $\beta_i, \ldots, \beta_k$ are all
     the constants in A;
     compute $Q = P \times \{t\}$;
     return $\xi_{\nu(\beta_1), \nu(\beta_2)}(Q)$;
  3: form a tuple $t = <\beta_i, \ldots, \beta_k>$, $1 \leq i \leq k \leq 4$, $\beta_i, \ldots, \beta_k$ are all
     the constants in A;
     compute $Q = P \times \{t\}$;
     return $\xi_{\nu(\beta_1), \nu(\beta_3), \nu(\beta_4); \nu(\beta_2)}(Q)$
  end

The algorithm starts off with an initial call to expand_goal with the two input parameters of a SchemaLog goal G and a relation M that provides the initial bindings for the variables of G.

3.5 Example of Rule-Goal Tree Expansion in SchemaLog

We now give an illustration of what a Rule-Goal Tree looks like when applied to a SchemaLog program during top-down computation.

Example 3.2 Let us consider the following SchemaLog program: (from example 2.1)

$A[T : quarter \rightarrow Q, ticker \rightarrow T, A' \rightarrow V', \text{comp.perf} \rightarrow V] \leftarrow \text{stocks}[ticker \rightarrow T, A' \rightarrow V'], A' \neq \text{ticker}, T[quarter \rightarrow Q, A \rightarrow V], A \neq \text{quarter}$
\[ A[X : \text{quarter} \rightarrow Q, \text{ind perf} \rightarrow V] \leftarrow A[X : \text{industry} \rightarrow B], \]
\[ \text{ind review}[\text{quarter} \rightarrow Q, \text{industry} \rightarrow B, A \rightarrow V], A \neq \text{quarter}, \]
\[ A \neq \text{industry} \]

with the query:
\[ ?- \text{aeps}[T : X \rightarrow Y]. \]

The following is the atomized version of this program:
\[ A[T : \text{quarter} \rightarrow Q] \leftarrow \text{body}_1 \]
\[ A[T : \text{ticker} \rightarrow T] \leftarrow \text{body}_1 \]
\[ A[T : A' \rightarrow V'] \leftarrow \text{body}_1 \]
\[ A[T : \text{comp perf} \rightarrow V] \leftarrow \text{body}_1 \]
\[ A[X : \text{quarter} \rightarrow Q] \leftarrow \text{body}_2 \]
\[ A[X : \text{ind perf} \rightarrow V] \leftarrow \text{body}_2 \]

where,
\[ \text{body}_1 \equiv \text{stocks}[T_1 : \text{ticker} \rightarrow T], \text{stocks}[T_1 : A' \rightarrow V'], A' \neq \text{ticker}, \]
\[ T[T_2 : \text{quarter} \rightarrow Q], T[T_2 : A \rightarrow V], A \neq \text{quarter} \]

and,
\[ \text{body}_2 \equiv A[X : \text{industry} \rightarrow B], \text{ind review}[T_1 : \text{quarter} \rightarrow Q], \text{ind review}[T_1 : \text{industry} \rightarrow B], \text{ind review}[T_1 : A \rightarrow V], A \neq \text{quarter}, A \neq \text{industry} \]

By factoring out rule bodies into temporary predicates \text{body}_i, re-computation is avoided. Figure 3.1 shows the rule/goal tree for this example. For lack of space, only two representative branches are shown in the figure.

Execution of our top-down algorithm invokes the \( \gamma \) operation of the algebra for the \text{expand_goal}() call corresponding to the subgoal \text{stocks}[T1 : A'\rightarrow V1] in Level 1. Call to \text{expand_rule}() for this rule invokes the \( \varphi \) operation. In fact, the non-classical algebraic operations (of type 2 or 3) are invoked at all the nodes depicted in this example tree.
3.6 Conclusion

In this chapter, we saw how a given *SchemaLog* program can be processed in a top-down manner in procedures similar to the classical top-down processing of first order logic programs. To accommodate the higher order of the *SchemaLog* syntax, we need to modify such well-known algorithms such as “unification”, “atomization”, “atov”, “vtoa”, etc. The *SchemaLog* version of “unification” was addressed in [Sub97]. We presented here the other algorithms mentioned above. The Rule-Goal Tree structure, which is basic to the top-down processing paradigm, was also illustrated with an example of a *SchemaLog* program.
Chapter 4

Physical Storage Architectures

4.1 Introduction

As illustrated in the preceding chapters, SchemaLog possesses powerful capabilities for querying data and meta-data of relations as well as for restructuring them. Supporting these features in an implementation requires efficient storage structures. Recall that SchemaLog is implemented by compiling its constructs into appropriate operations in Schema Algebra (SA). Clearly, depending on the chosen storage structures, the underlying implementation strategy for the algebraic operations would differ.

In this chapter, we outline three alternative storage structures at the level of physical schemas. We also discuss the implementation of SA operators (both conventional and additional) corresponding to each of them. After each algorithm we discuss the cost of implementing the algorithm in terms of tuple reads (or writes). We use tuple reads as a theoretical way of comparing the costs among the various strategies. The actual time taken for the various operations would be platform-dependent and will vary according to such factors as disk space, main memory size, block size, clock speed, etc. In a later chapter we shall also discuss a practical implementation of these algorithms in MS-Access and compare the actual costs of the operations under the various strategies.

Before we present the alternative storage structures, we remark that for existing (i.e. base) database relations, it is unrealistic to suppose that they can be converted into any form other than their existing form. For one thing, such a conversion would incur a massive overhead. For another, this would disrupt applications running on
the existing database. [LSS96] discusses these issues in detail and argues that from a practical perspective, the base relations should be preserved in their existing form. Thus, we are really considering alternative storage structures for database relations which are created or derived by SchemaLog programs.

4.2 Conventional Storage

The idea is to use the same schema as for conventional database relations. In other words, a relation \( r \) with attributes \( A, B, C \) would be implemented as a file of records with those fields. Thus, derived database relations would be stored and accessed the same way as base relations are.

We now present algorithms to implement some chosen operators.

4.2.1 Selection

We consider two principal cases: one in which the selection condition(s) involve(s) only key attribute(s), and the other in which non-key attributes are also involved. In the first case, we consider two possibilities: one in which there exists an index on the key attribute(s), and the other in which no index exists. Assume that the selection is done on a relation called \( r \).

Algorithm 4.1 Selection on indexed key attribute(s)

\[
\begin{align*}
\text{Algorithm 4.1} & \quad \text{Selection on indexed key attribute(s)} \\
\text{begin} & \\
\text{result} & = \phi \\
\text{use the index on the key attribute to read the relevant tuple(s)} & \\
\text{result} & = \text{result} \cup \text{tuple(s)} \\
\text{write result} & \\
\text{end}
\end{align*}
\]

Cost

In the worst case, the whole relation may need to be read and written once (ie, if all tuples are selected). For uniformity of comparison, we take the most common case of selection, with the equality operator. Here, we read and write at most one tuple.
Next, we take up the case when there is no index on the key attribute. We make the assumption that the file is ordered on the key attribute.

\textbf{Algorithm 4.2} Selection on non-indexed key attribute(s)
\begin{itemize}
  \item \texttt{result} = \emptyset
  \item use binary search to locate the tuple(s) satisfying the given condition
  \item if such tuples are found, \texttt{result} = \texttt{result} \cup \texttt{tuple(s)}
  \item write \texttt{result}
\end{itemize}
\end{algorithm}

\underline{Cost}

The reading costs $\log_2 N_t$. Here (and in the following pages), $N_t$ stands for the “average number of tuples in a relation”. The expression stands for the number of tuples read during a binary search of the file containing the relation. The writing involves at most one tuple (assuming equality selection).

The final case of selection we consider involves non-key attributes for which no indexes exist.

\textbf{Algorithm 4.3} Selection on non-key attribute(s)
\begin{itemize}
  \item \texttt{result} = \emptyset
  \item \texttt{while} there are tuples in \texttt{r}
    \item \texttt{read a tuple}
    \item \texttt{check whether it satisfies the given condition}
    \item if it does, \texttt{result} = \texttt{result} \cup \texttt{tuple}
  \item \texttt{endwhile}
  \item write \texttt{result}
\end{itemize}
\end{algorithm}

\underline{Cost}

The reading costs $N_t$ (linear read of all tuples in the relation); the writing costs $S_d$. $S_d$ stands for “selection density”. Selection Density for a specified column is
defined as the average number of times a specific value occurs in that column. The number of distinct values in that column would then be $N_t/S_d$.

Note that if secondary indexes are available, selection on non-key attributes can be further optimized.

### 4.2.2 Projection

Projection involves two cases: one in which duplicates are not eliminated and the other in which they are.

---

**Algorithm 4.4 Projection without elimination of duplicates**

begin

result = φ

while there are tuples in r

read a tuple

remove unwanted attributes from the tuple

result = result ∪ tuple

endwhile

write result

end

---

**Cost**

The reading costs $N_t$; and the writing, too, costs $N_t$.

---

**Algorithm 4.5 Projection with elimination of duplicates**

begin

read in the relation $r$

sort $r$ using the desired attribute(s) as key

write back $r'$ without duplicates and without the unwanted attributes

end

---

**Cost**

If the file is small enough to fit into main memory, the cost can be approximated as $(N_t + N'_t)$ where $N'_t$ is the number of tuples in a relation after duplicate elimination. It can be considered equal to $N_t/S_d$. 

37
For files larger than main memory, an external sort is required; the cost approximates to $4N_t$.

### 4.2.3 Join

We consider two-way joins only, of relations $r_1$ and $r_2$. And in the most general case, we assume that no indexes are available. In the first case considered, the join attribute(s) are a key of both $r_1$ and $r_2$. Both relations are ordered on the key.

---

#### Algorithm 4.6 Join on Key attributes

begin

`result = ∅`

*Let $P_1$ point to the first tuple of $r_1$*

*Let $P_2$ point to the first tuple of $r_2$*

while there remain tuples in $r_1$ and $r_2$

  if the join attribute values in the two tuples match,

    make the new join tuple $t$

    `result = result ∪ t`

    move $P_1$ and $P_2$ down their respective relations

  elseif $P_1$ value $<$ $P_2$

    move $P_1$ down by a row

  elseif $P_1$ value $>$ $P_2$

    move $P_2$ down by a row

endif

endwhile

end

---

#### Cost

The reading costs $2 \times N_t$ since each relation is read once. The writing costs $N_t \times J_d$, where $J_d$ stands for "Join Density". Join Density is defined here as the ratio of the number of tuples produced by the join to the number of tuples in a relation. It follows that the number of tuples produced by the join is $N_t \times J_d$. 

---

38
When the join attributes are not keys (and no indexes are available), the most efficient method for the join, in most cases, is to first sort the two relations on the join attribute(s) and then to make the join. This is traditionally known as a Sort Join.

**Algorithm 4.7 Sort Join**

begin
    sort \( r_1 \) on the join attribute(s)
    sort \( r_2 \) on the join attribute(s)
    make the join as in Algorithm 4.6 above
end

**Cost**

We assume that each sort of a relation involves at least one read and one write of all the tuples in the relation, making up a cost of \( 2 \times N_t \). The total cost would, then, be \( 2N_t + 2N_t + (2N_t + N_t \times J_d) \).

Let us now consider the case when the join attributes have secondary indexes. We look at the join \( r_1 \rightarrow r_1.A = r_2.B \rightarrow r_2 \).

**Algorithm 4.8 Indexed Join**

begin
    repeat
        read a block of \( r_1 \) tuples with the same values on \( A \)
        read the corresponding tuples in \( r_2 \) (ie, with the same values for \( B \))
        make the join and add the tuples to the result
    until there are no more tuples in \( r_1 \)
end

**Cost**

The cost comes to \( 2N_t + N_t \times J_d \).

For more detailed discussions of Join strategies and cost computations, one can refer to any standard Text on Databases such as [Des90], [KS91], [Ram97], [Ull89].
4.2.4 Fetching Relation Names

We assume that the DBMS maintains the names of all its relations in its system tables. The application of the \( \rho \) operator, then, involves one read through the corresponding system table and a write of all relation names.

The cost would be \( N_r \times N_a + N_r \), where \( N_r \) is defined as the number of relations in the database and \( N_a \) as the average number of attributes in a relation.

4.2.5 Fetching Relations and their Schemas

Here again, we assume that the information on relations and their attributes is available in the form of system tables. The schema of a system table is generally of the form \((\text{relation, attribute})\). The application of the \( \alpha \) operator involves using the relation name (supplied as an argument to the operator) to search the system tables and write out tuples of the form \((\text{relation, attribute})\) for each attribute in the relation.

The cost of this operation would be \( N_r \times N_a \) for reading and \( N_r \times N_a \) for writing.

4.2.6 Querying Data and Meta-data

Algorithm 4.9 \( \gamma_{ip}(s) \) Querying with a single pattern

begin
 Identify the relations involved by inspecting the \( i^{th} \) column of relation \( s \).
 for each relation \( r \) thus identified
 for each tuple \( t \) in \( r \)
     if \( t \) matches the pattern \( p \), add to the result a tuple
         of the form \(< r, \text{tid}, a, v >\) where \text{tid} is the tuple-id of
         \( t \) and \( (a,v) \) is an associated witness pair
     endfor
 endfor
end
Cost

The reading cost can be written as $N_t \times N'_r$, where $N'_r$ denotes the number of relations involved in the operation.

Note that this is the worst-case scenario. The cost can be considerably reduced if the pattern is of the form $\langle a\rightarrow \rangle$ or $\langle a\rightarrow v \rangle$. In such cases, during the first step of the algorithm above, we choose only relations with the attribute $a$ in their schemas. Note also that if there exists an index or an ordering on $a$, the algorithm can be further optimized.

In the worst case, i.e., if the pattern is $\langle \rightarrow \rangle$, we have the cost as $N_t \times N_a \times N'_r$. Writing costs for the other patterns will vary widely depending on whether $a$ and/or $v$ are known and what their actual values are. For a rough estimate, we use a factor called "Pattern Density (Single)" or $P_{ds}$. Pattern Density, in general, is defined as the number of tuples that actually fit the pattern (whether single or multiple) to the maximum number of tuples that could be formed. With this factor, then, we can summarise the writing cost as $N_t \times N_a \times N'_r \times P_{ds}$.

---

Algorithm 4.10 $\gamma^\wedge_{s_i(p_1,\ldots,p_n)}(s)$ Querying with a multiple pattern

begin

Identify the relations involved by inspecting the $i^{th}$ column of relation $s$.

for each relation $r$ thus identified

for each tuple $t$ in $r$

if $t$ matches the pattern $(p_1,\ldots,p_n)$,

add to the result as many tuples as possible of the form $(r,a_1,v_1,\ldots,a_n,v_n)$ using $(a,v)$ combinations of tuple $t$

in all ways that fit the pattern

endfor

endfor

end

---

Cost

The reading cost is the same as that for the single pattern query, i.e., $N_t \times N'_r$.

Here, too, the writing cost will vary widely according the number of patterns in the condition, the number of attributes and other such factors. In cases where the patterns do not contain any (or few) attribute names, each original tuple could be
used to compose a number of tuples in the output. We use the factor “Pattern Density (Multiple)” or \( P_{dm} \) to help us arrive at an expression for the cost. “Pattern Density” has already been defined earlier. If each original tuple is used to make at most one tuple in the output, the number of tuples written would be \( N_t \times N_r' \times P_{dm} \). If we assume on the average that a tuple could be used to make \( N_a \) tuples in the output, the writing cost would be \( N_t \times N_r' \times N_a \times P_{dm} \).

### 4.2.7 Creating Relations

**Algorithm 4.11 \( \kappa_i(s) \)**

```
begin
    while there are rows to be considered in s (the given schemaless relation)
        read a row
        store the relation name from the \( i^{th} \) column of s
    endwhile
    for each relation name r stored
        create a table called r
    endfor
end
```

**Cost**

The reading cost is \( N_t \) (if the “schemaless” relation is also considered to have the average \( N_t \) number of rows). This cost is negligible compared to the cost of creating the tables which is obviously much larger.

The writing cost can be estimated to be \( N_r \times T_c \), where \( N_r \) is the number of relations identified by \( \kappa \) and \( T_c \) is the cost of creating a table.

### 4.2.8 Creating Relations with Schemas

**Algorithm 4.12 \( \zeta_{ij}(s) \)**

```
begin
    while there are rows to be considered in s (the given schemaless relation)
        read a row
```

42
store the relation name from the $i^{th}$ column and the attribute name from
the $j^{th}$ column of $s$ in an appropriate data structure

endwhile

for each relation name $r$ stored

create a table called $r$ with the corresponding stored attributes

endfor

end

Cost

The reading and writing costs can be considered identical to that of the previous
operator, $\kappa$.

4.2.9 Creating Relations with Schema and Data

In this operation, information about the schema of a table as well as about the data
contained in it arrive in a piecemeal fashion. Data can be added piecemeal to the
conventional relational storage. But the modification of schema is an expensive op-
eration and impractical to do "on the fly" as soon as a new column name appears
as input. In our algorithm below, all information about tables (old as well as new),
schema and data available in the "schemaless relation" (output by the computation
of a SchemaLog program) is stored in temporary data structures and used to make
temporary tables. When all expected information is in, these temporary tables are
joined (wherever possible) to existing database tables.

Algorithm 4.13 $\mathfrak{g}_{i,j,k;r}(s)$

begin

while there are rows to be considered in $s$ (the given schemaless relation)

read a row

store the relation name, attribute name, value and grouping

attribute value in an appropriate data structure

endwhile

for each relation name rel in the stored relation

create a table called temp_rel with the appropriate (stored) attributes
append tuples to the table (using the stored values)
endfor
for each temp_rel created
if ∃ a relation named rel in the database
   rel = rel ÷ temp_rel
else create a relation rel = temp_rel
endfor
end

Cost
The cost (reading as well as writing) can be separated into the following parts:

- reading in the Schemaless relation = \( N'_r \times N_a \times N_t \)
- creating temporary data structures = \( N'_r \times N_a \times N_t \)
- creating temporary tables = \( N'_r \times T_c \)
- writing to the temporary tables = \( N'_r \times N_t \)
- making joins with existing tables = \( N'_r \times 'joincost' \)

We have made use of previously defined terms. \( N'_r \) stands for the average number of tables that would be mentioned in the input.

4.3 Reduced Storage

The term reduced refers to the fact that SchemaLog admits a faithful first-order reduction, as established in [LSS96]. The idea is that each relation in the database can be “flattened” and all the information in the database can be compiled into three relations – \( call_4(R, T, A, V) \) (corresponding to \( R[T : A→V] \)), \( call_2(R, A) \) (corresponding to \( R[A] \)), and \( call_1(R) \) (corresponding to \( R[] \)).

For example, consider Figure 4.1 which shows two tables of conventional data storage.

In the “Reduced Storage”, this information would take the form of the tables given in Figure 4.2

As pointed out earlier, in a practical setting, this flattening can only be applied to “derived” database relations. Under reduced storage, meta-data querying is essentially reduced to conventional data querying, and restructuring is reduced to updating the \( call_i \) relations. In other words, the extended operations of type (2) and (3) in
<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>35</td>
<td>Montreal</td>
</tr>
<tr>
<td>Peter</td>
<td>42</td>
<td>Toronto</td>
</tr>
<tr>
<td>Andrew</td>
<td>31</td>
<td>Montreal</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>company</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>CIBC</td>
</tr>
<tr>
<td>Peter</td>
<td>IBM</td>
</tr>
</tbody>
</table>

Figure 4.1: Conventional Storage Tables

$S.A$ are reduced to classical RA operations, under this storage. In particular, the piecemeal computation of the operator $\pi$ under conventional storage reduces to normal computation (where entire tuples are computed at a time, rather than in parts). By contrast, simple classical operations like selection and join translate into complex operations in this storage method.

We now provide algorithms to implement the various Relational Algebraic Operators we had mentioned with regard to the Conventional Strategy. In the algorithms and in the cost analysis which follow, we shall assume the presence of three indexes: a primary key index on $(relation, tid)$ and a secondary key indexes on attribute and value. Considering that the whole database in this strategy consists practically of one relation, the call$_4$ relation, a total of three indexes is a very reasonable overhead.

### 4.3.1 Selection

Algorithm 4.14 $\sigma_{a=v}r$

begin

scan the call$_4$ relation using the index on value
and write out the tuple set, $S$, with the value $v$, attribute $a$ and relation $r$

result = \emptyset

for each tuple in the set $S$

scan the call$_4$ relation for all tuples, $P$, in $r$
with the same tid (using the (relation, tid) index)

result = result $\cup$ $P$

endfor

end
Cost

Producing the tuple set $S$ involves $S_d \times N_r$ tuple reads and $S_d$ tuple writes. There is a further read of $S_d$ tuples in the for loop and the final write of $S_d \times N_a$ tuples in the result.

### 4.3.2 Projection

**Algorithm 4.15** $\Pi_{a_1, \ldots, a_n} r$

begin
   
   \[ \text{result} = \emptyset \]
   
   scan the \text{call}_4 relation using the index on \text{r}

\[ 46 \]
for each tuple t thus retrieved
    if the attribute is in the set \{a_1, \ldots, a_n\}
        \text{then, result} = \text{result} \cup t
    endfor
end

Cost

The algorithm involves a read of \(N_a \times N_t\) tuples (i.e., the subset of the call relation corresponding to the relation \(r\), and a write of \(n \times N_t\) result tuples, where \(n\) is the number of attributes projected.

4.3.3 Join

When relation tuples are stored in the Reduced Storage strategy, the joinable tuples have to be first located, and their \(tid\)'s used to locate the other reduced tuples that form part of the conventional joined tuple. The basic outline of the algorithm is given below.

---

Algorithm 4.16 \(r_1 \bowtie r_2\)

begin
    \text{result} = \emptyset
    create a temporary relation temp_rel with the scheme \((r_1, tid_1, r_2, tid_2)\)
    use a conventional join algorithm on two copies of the call relation
to match tuples where the relation names and the attribute names correspond to the join requirement and the values are the same.
write out tuples of the form \((r_1, tid_1, r_2, tid_2)\) into temp_rel
    for each row \(t\) in temp_rel
        retrieve all rows in call relation with the same values for \(r_1\) and \(tid_1\)
        for every row \(r\) thus retrieved
            assign the new relation name (ie, the name of the joined table) to the 1\textsuperscript{st} column
        assign a tuple id
        assign a new column name (if desired)
/* the contents of the value column are left the same*/
result = result ∪ r
endfor

retrieve all rows in call₄ with the same values for r₂ and tid₂
for every row s thus retrieved
    assign the new relation name (ie, the name of the joined table)
    to the 1st column
    assign the tuple id /* same as for r above */
    assign a new column name (if desired)
    /* the contents of the value column are left the same*/
    result = result ∪ s
endfor
endfor
end

---

Cost

First we consider the cost of the join on the two copies of call₄. Using the Indexed Join, this would be $2 \times N_a \times N_t + N_t \times N_a \times J_d$. At this stage the temp_rel mentioned in the algorithm is estimated to contain about $N_t \times N_a \times J_d$ tuples. In the remaining part of the algorithm, for each row in temp_rel, the two for loops read and write exactly $N_a$ tuples each. The cost for this part would then be $2 \times N_t \times N_a \times J_d \times N_a$ for the read and $2 \times N_t \times N_a \times J_d \times N_a$ for the write.

### 4.3.4 Fetching Relation Names

In the Reduced Storage Strategy, the output of the $\rho$ operator is simply the stored call₁ relation.

The number of rows in call₁ usually very small and the cost the operation can be considered negligible.

### 4.3.5 Fetching Relations and their Schemas

This operation, too, is somewhat trivial under this Storage scheme. An implementation of the operation $\alpha_i(r)$ would simply produce a subset of the tuples in the call₂
relation.

The cost can be estimated as $N_r \times N_a$ for reading and $N_r \times N_a$ for writing.

4.3.6 Querying Data and Meta-data

Querying with a single pattern is fairly straightforward. We only have to read out tuples from $call_4$ corresponding to the relations identified by the operator and check whether the tuples match the pattern.

Algorithm 4.17 $\gamma_{i,p}(s)$ Querying with a single pattern

begin

Identify the relations involved by inspecting the $i^{th}$ column of relation $s$.

for each relation $r$ thus identified

for each tuple $t$ in $r$

if $t$ matches the pattern $p$, add $t$ to the result

dendif

dendif

dend

end

Cost

The reading cost can be written as $N_t \times N_a \times N'_r$, where $N'_r$ denotes the number of relations identified by the operator. This, again, is a basic and general algorithm. It can be optimized according the nature of the pattern. If both attribute and value are known, for example, the indexes on attribute and value can be used to go straight to the tuples matching the pattern and then the relation names can be checked against the required relation names.

Making use of the $P_{ds}$ factor (defined earlier), we can summarise the writing cost as $N_t \times N_a \times N'_r \times P_{ds}$.

Querying for a multiple pattern is somewhat complex in this Strategy because of the very nature of the storage method. Each row in $call_4$ contains information on a
single pattern only; hence, rows with the same tid have to be put together before we can decide on the occurrence or otherwise of the given multiple pattern.

Another point to note here is that a single conventional row can produce a number of rows in the output in cases where one or more of the attribute positions are left blank in the pattern. For example, if \( \langle \text{John}, 35, \text{Montreal} \rangle \) is a tuple in the relation \( \text{person}(\text{name}, \text{age}, \text{city}) \) and the given pattern is \( \langle \rightarrow, \rightarrow \rangle \), then the \( \wedge \) operator would produce the following three tuples: \( \langle \text{person}, \text{name}, \text{John}, \text{age}, 35 \rangle \), \( \langle \text{person}, \text{name}, \text{John}, \text{city}, \text{Montreal} \rangle \) and \( \langle \text{person}, \text{age}, 35, \text{city}, \text{Montreal} \rangle \). In general, if there are \( N_a \) attributes and \( p \) patterns, a single conventional tuple can produce \( N_a C_p \) tuples in the output.

The algorithm below does not go into the details of implementation for each type of pattern. It outlines a general procedure applicable in all cases.

---

**Algorithm 4.18 \( \wedge_{i\in\{p_1,\ldots,p_n\}}(s) \)**

*Querying with a multiple pattern*

**begin**

*Identify the relations involved by inspecting the \( i \)th column of relation \( s \).*

*for each relation \( r \) thus identified*

*for each set \( R \) of tuples with the same tid*

*if the tuples in \( R \) match the pattern \( \langle p_1, \ldots, p_n \rangle \),*

*add to the result as many tuples as possible of the form*

\( \langle r, a_1, v_1, \ldots, a_n, v_n \rangle \) using \( (a,v) \) combinations of the tuple set \( R \) in all ways that fit the pattern

*endfor*

*endfor*

**end**

---

**Cost**

The reading cost is the same as that for the single pattern query, i.e., \( N_t \times N_a \times N_r' \).

If each tuple set \( R \) (see algorithm above) is used to make at most one tuple in the output, the number of tuples written would be \( N_t \times N_r' \times P_{dm} \), where \( P_{dm} \) is the "Pattern Density" factor for a multiple pattern. If we assume on the average that set \( R \) could be used to make \( N_a \) tuples in the output, the writing cost would be \( N_t \times N_r' \times N_a \times P_{dm} \).
4.3.7 Creating Relations

The effect of the \( \kappa \) operator in this Strategy is to add as many new rows to the \( \text{call}_1 \) relation as there are relations to be created. The cost is negligible.

4.3.8 Creating Relations with Schemas

Algorithm 4.19 \( c_{i,j}(s) \)

begin

while there are rows to be considered in \( s \) (the given schemaless relation)
read a row
store the relation name from the \( i^{th} \) column and the attribute name from
the \( j^{th} \) column of \( s \) in an appropriate data structure
endwhile

for each relation name \( r \) stored

for each attribute \( a \) in \( r \)

insert a row \( \langle r, a \rangle \) in the \( \text{call}_2 \) relation

endfor

insert a row \( \langle r \rangle \) in the \( \text{call}_1 \) relation

endfor

end

Cost

\( N'_r \times N_a \) rows are written to the \( \text{call}_2 \) relation; and \( N'_r \) rows are written to the \( \text{call}_1 \) relation.

4.3.9 Creating Relations with Schema and Data

This operation which was quite complex in the Conventional Strategy is quite straightforward under this Strategy. The given "Schemaless relation" is already in the same format as the \( \text{call}_4 \) relation. And so we only need to take each row in the input, rearrange the column values as specified in the operator and insert the row into \( \text{call}_4 \).
Algorithm 4.20 $\beta_{i,j,k,t}(r)$

begin
    while tuples remain in $r$
        read tuple $\tau$ from $r$.
        form a tuple $<r,t,a,v>$ such that $r$, $t$, $a$, and $v$ are the $i^{th}$, $\ell^{th}$, $j^{th}$, and $k^{th}$ components respectively of $\tau$.
        write the new tuple.
    endwhile
end

Cost
Reading in the input relation needs $N'_r \times N_a \times N_t$ tuple reads; and the same number of tuples are written to the call$_4$ relation.

4.4 Reduced, Atomized Storage

This is really a refinement on the previous storage scheme. Since information pertaining to several different database relations is lumped into one relation call$_4$ (and also call$_2$, call$_1$) there is some attendant redundancy e.g., in the repetition of relation names. One way to minimize this redundancy is to (i) separate information in different database relations in their flattened representations, and (ii) split the information corresponding to different attributes of the same relation, using the tuple id's as a glue.

A derived database relation of the form $r(a_1,\ldots,a_n)$ is physically stored in relations physrel$(r,a_1)(tid,val),\ldots,$ physrel$(r,a_n)(tid,val)$. Note that in this storage scheme, physrel$(r,a_j)$ is the name of a relation used for physical storage while $\{tid,val\}$ is its schema. The first column in a physical relation corresponds to the tuple-ids of tuples in the database relation $r$, and the second column contains the values. The tuple $<i,v>$ in a physical relation physrel$(r,a_j)$ represents the fact that a tuple $i$ in relation $r$ has value $v$ on attribute $a_j$.

To illustrate the above, let us consider the data in the conventional tables of Figure 4.1. In the “Reduced, Atomized Storage”, this information would take the form of the tables given in Figure 4.3
\begin{figure}
\centering
\begin{tabular}{|c|c|}
\hline
\text{tid} & \text{value} \\
\hline
\text{t}_1 & John \\
\text{t}_2 & Peter \\
\text{t}_3 & Andrew \\
\hline
\end{tabular}
\caption{\textit{physrel}(\textit{person}, \textit{name})}

\begin{tabular}{|c|c|}
\hline
\text{tid} & \text{value} \\
\hline
\text{t}_1 & 35 \\
\text{t}_2 & 42 \\
\text{t}_3 & 31 \\
\hline
\end{tabular}
\caption{\textit{physrel}(\textit{person}, \textit{age})}

\begin{tabular}{|c|c|}
\hline
\text{tid} & \text{value} \\
\hline
\text{t}_1 & Montreal \\
\text{t}_2 & Toronto \\
\text{t}_3 & Montreal \\
\hline
\end{tabular}
\caption{\textit{physrel}(\textit{person}, \textit{city})}

\begin{tabular}{|c|c|}
\hline
\text{tid} & \text{value} \\
\hline
\text{t}_1 & John \\
\text{t}_2 & Peter \\
\hline
\end{tabular}
\caption{\textit{physrel}(\textit{works\_in\_name})}

\begin{tabular}{|c|c|}
\hline
\text{tid} & \text{value} \\
\hline
\text{t}_1 & CIBC \\
\text{t}_2 & IBM \\
\hline
\end{tabular}
\caption{\textit{physrel}(\textit{works\_in\_company})}
\end{figure}

Thus, in this strategy, a derived database relation is stored using as many physical relations as there are attributes in it – each such relation storing one column of the database relation. The relation name and the attribute name corresponding to the column are ‘encoded’ in the name of the physical relation. The $SA$ operations are interpreted against such a representation; for instance, operations that add attributes to an existing relation translate in this strategy to operations that add new relations. Many of the comments made for reduced storage also apply to reduced atomized storage. Thus, we expect that this scheme will suit meta-data querying and restructuring better than conventional data querying.
We now proceed to the algorithms for implementing Algebraic Operators under this Strategy. Here we shall assume that each physrel relation is indexed on tid and on value.

4.4.1 Selection

Algorithm 4.21 $\sigma_{a=v}r$

begin

for each tuple in physrel($r, a$)

read tuple $t$

if $t.val = v$, write $t$ to the output relation physrel($r_{out}, a$)

for each physrel($r, a_i$), $a_i \neq a$

use the index on tid to read from physrel($r, a_i$) the tuple with tid $t.tid$

write this tuple to the output relation physrel($r_{out}, a_i$)

endfor

endfor

end

Cost

If $S_d$ is the "Selection Density" (defined earlier), the outer for loop executes $S_d$ times. And the inner loop executes $N_a$ times (actually $N_a - 1$ times; but remember that $N_a$ is considered an average value). There are therefore $S_d \times N_a$ tuple reads and the same number of tuple writes.

4.4.2 Projection

If the projected attributes contain key attributes or if duplicate elimination is not required, the projection operation is very simple in this strategy as it would involve merely marking the appropriate physrel's.

The algorithm below assumes that duplicate elimination is required in the result of the projection.
Algorithm 4.22 $\Pi_{a_1, \ldots, a_n}(r)$
begin
make a join on tid among all $\text{physrel}(r, a_i)$ where $i = 1, \ldots, n$, retaining only the value columns.
sort the resultant relation
write out the same relation eliminating duplicates
write out the $n$ columns from this relation into $n$ physrel relations
assigning new tid's for each row
end

Cost
The cost of the Join ("indexed join") is $2N_t \times J_d \times N_t$. Sorting and rewriting involves approximately $2N_t + N'_t$ tuple reads and writes. Writing the result into physrel schemes costs $n \times N'_t$ where $N'_t$ stands for the number of tuples after duplicate elimination and $n$ represents the number of attributes projected.

4.4.3 Join
The complexity of a Join operation in this Strategy arises from the fact that information on a single "conventional tuple" is scattered across several physical relations. For a simple join such as $r \bowtie_{m=n} s$, first $\text{physrel}(r, m)$ and $\text{physrel}(s, n)$ are joined in the conventional way. Thereafter the other columns in the "join" tuple will have to be assembled one by one using the tid's of $r$ and $s$ as glue. The assembled "join" tuple will also have to be assigned a new tuple id. In the algorithm below a function $\text{newtid}(i, j)$ that generates a unique tid based on tid's $i$ and $j$ is used.

Algorithm 4.23 $r \bowtie_{m=n} s$
begin
make a join on value between $\text{physrel}(r, m)$ and $\text{physrel}(s, n)$
producing tuples of the form \((r[tid], r[val], s[tid], s[val])\)

for each row obtained above

for each relation physrel\((r, a_i)\)

read tuple \(t_r\) with tid \(r[tid]\)

write \((\text{newtid}(r[tid], s[tid]), t_r[value])\) in physrel\((\text{join}, a_i)\)

endfor

for each relation physrel\((s, a_j)\)

read tuple \(t_s\) with tid \(s[tid]\)

write \((\text{newtid}(r[tid], s[tid]), t_s[value])\) in physrel\((\text{join}, a_j)\)

endfor

endfor

define

Cost

The cost of the first join is \(2N_t + J_d \times N_t\). For each row produced by join, we have two \textit{for} loops, each loop consisting of one tuple read and one tuple write, and iterating on the average \(N_a\) times. The cost for this part, then, would be \(J_d \times N_t \times 4N_a\).

4.4.4 Fetching Relation Names

Here we assume the existence of system tables (as in the "Conventional" Strategy). The cost would be \(N_r \times N_a\) tuple reads and \(N_r\) tuple writes.

4.4.5 Fetching Relations and their Schemas

The implementation of the operation \(\alpha_i(r)\) would also refer to the system tables. The cost can be estimated as \(N_r \times N_a\) for reading and \(N_r \times N_a\) for writing.

4.4.6 Querying Data and Meta-data

In the Single Pattern query, if the attribute name is given in the pattern, we go straight to only those physrel relations relevant to the given relation and attribute, physrel\((r, a)\). Otherwise, every physrel relation with the given relation name has to
be queried against the pattern. And if the value is given in the pattern, the index on value can be used to speed up the operation.

---

**Algorithm 4.24** $\mathcal{g}_{ip}(s)$ *Querying with a single pattern*

begin

*Identify the relations involved by inspecting the $i^{th}$ column of relation $s$.*

for each relation $r$ thus identified

for each physical relation $\text{physrel}(r, a_i)$

read each tuple $t$

if $t$ matches the pattern $p$, form an output tuple of the form $\langle r, \text{tid}, a_i, \text{value} \rangle$, where $r$ and $a_i$ are taken from the current physrel and tid, value are taken from the current tuple $t$

endfor

endfor

end

---

**Cost**

In the most general case, i.e., if the pattern is $\langle \Rightarrow \rangle$, the reading cost can be written as $N_t \times N_a \times N'_r$, where $N'_r$ denotes the number of relations identified by the operator.

Making use of the $P_{ds}$ factor, we can summarise the writing cost as $N_t \times N_a \times N'_r \times P_{ds}$.

Querying for a multiple pattern introduces similar complexities here as in the *Reduced Strategy*. Each physrel contains information on a single pattern only; hence, rows with the same tid from different physrel's have to be put together before we can decide on the occurrence or otherwise of the given multiple pattern. The algorithm below outlines a general procedure applicable in all types of patterns.
Algorithm 4.25 $\gamma_{i(p_1,\ldots,p_n)}(s)$ \hspace{1cm} Querying with a multiple pattern

begin

Identify the relations involved by inspecting the $i^{th}$ column of relation $s$.
for each relation $r$ thus identified

select a physrel of the relation

for each tuple $t$ in the physrel

select tuples with the same tid from the

remaining physrel's of the relation

if the $(a,v)$ combinations of this set of tuples

match the pattern $(p_1,\ldots,p_n)$,

add to the result as many tuples as possible of the form

$(r,a_1,v_1,\ldots,a_n,v_n)$ using $(a,v)$ combinations from the

tuple set under consideration in all ways that fit the pattern

done

done

end

Cost

The reading cost is $N_t \times N_a \times N_r'$ (same as that for the single pattern query).

If each "conventional" tuple (obtained by placing tuples with the same $tid$ from
physrel's of the same relation side by side) is used to make at most one tuple in
the output, the number of tuples written would be $N_t \times N_r' \times P_{dm}$, where $P_{dm}$ is the
"Pattern Density" factor for a multiple pattern. If we assume on the average that
such a "conventional" tuple can be used to make $N_a$ tuples in the output, the writing
cost would be $N_t \times N_r' \times N_a \times P_{dm}$.

4.4.7 Creating Relations

The $\kappa$ operator in this Strategy would merely add relation names to the System
Table. The cost is negligible.
4.4.8 Creating Relations with Schemas

Algorithm 4.26 $\varsigma_{i,j}(s)$

begin
    while there are rows to be considered in $s$ (the given schemaless relation)
        read a row
        store the relation name from the $i^{th}$ column and the attribute name from
        the $j^{th}$ column of $s$ in an appropriate data structure
    endwhile
    for each relation name $r$ stored
        for each attribute $a$ in $r$
            create a physical relation $\text{physrel}(r,a)$
            with attributes tid and value
        endfor
    endfor
end

Cost

$N_r' \times N_a$ physical relations are created. If $T_c$ is the cost of creating a table, the
total cost is $N_r' \times N_a \times T_c$.

4.4.9 Creating Relations with Schema and Data

In this Strategy, too, the $\sigma$ operator has a simple implementation. We look at each row
input by the given "Schemaless" relation, check whether the $\text{physrel}$ corresponding to
the relation and attribute exists and add the relevant tuple to the table.

Algorithm 4.27 $\varsigma_{i,j,k,l}(r)$

begin
    for each tuple in the input relation $r$
        read tuple $t$.
        if the table $\text{physrel}(t[i],t[j])$ does not exist
            create the table $\text{physrel}(t[i],t[j])$
            add tuple $< t[l],t[k] >$ to the relation $\text{physrel}(t[i],t[j])$
        endfor
end
Cost

Reading in the input relation needs \( N' \times N_a \times N_t \) tuple reads; and the same number of tuples are written to the various physrel relations. If we assume that on the average half of the \( N' \) relations are to be created, the cost of creating these tables would be \( \frac{1}{2} N' \times N_a \times T_c \).

4.5 Conclusion

From long theoretical study as well as long practical usage, it is clear that the storage of data in the conventional method of tables with well-defined schemes is well-suited to the efficient implementation of the traditional Relational Algebraic Operators. Since \( SA \) contains new operators both for querying as well as restructuring the database, the question is whether the conventional storage methods are still the best for the purpose of storing the data on which these operators are applied. We therefore presented in this chapter two new storage strategies – the "reduced" storage, and the "reduced, atomized" storage – which we consider might be more suitable for certain operations, especially the restructuring operations introduced in \( SA \). For all three strategies, we presented algorithms for the implementation of \( SA \) operators and estimated the theoretical cost.

If we consider the Join operator as representative of all querying operators and the \( e \) operator as representative of the restructuring operators and compare their theoretical costs across the three strategies, it is immediately evident that querying costs are low in the Conventional Strategy and very high in the other two strategies. As regards restructuring, the position is reversed: costs are high in the Conventional Strategy and much lower in the other two. Comparing the theoretical costs for the other querying and restructuring operations leads to similar conclusions. In applications which have a high proportion of queries and relatively few restructuring operations, the conventional storage method would therefore appear to be the better option. Where restructuring is frequent, one of the other two methods would appear to be the better choice.

In the next chapter, we shall analyse the experimental results from the implementation of the three strategies discussed here. We shall see there whether the
theoretical results agree with the experimental results. And in the case of restructuring operations in particular, the experimental results will help us to decide which of the three strategies is the most efficient.
Chapter 5

Experimental Results

5.1 Introduction

This chapter presents a report on the practical implementation of the $SA$ operators under the various storage strategies presented in the previous chapter. In that Chapter we also estimated the theoretical cost of the implementation of the $SA$ operators. To supplement those theoretical estimates, we study in this chapter the actual running time of various querying and restructuring operations on databases of varying sizes and other parameters such as Join Density and Selection Density. For each set of parameters, we present a table of readings taken for the various operations under the three strategies mentioned in the preceding chapter. Graphs on the table values illustrate the comparative suitability of the strategies for specific operations. We conclude the chapter with an analysis of the results obtained.

The implementation was done in the MS-Access Database Management System on the PC/Windows platform. All source code for the creation of tables for the test bed, the working of the $SA$ operators, etc. was written in Visual Basic.

5.2 Preparation of the Test Bed

The first task at the experimental stage was to prepare a set of tables (according to the three Strategies under consideration) with adjustable parameters such as the number of tuples per relation, the number of attributes per relation, the selection density of various columns and the join density between two columns. On the implementation
interface the user is prompted to supply the number of tables to be created, the number of columns in each table, the number of rows per table, the selection density of a selected column in a table and the join density between two selected columns in two tables. Making use of these parameters, the specified number of conventional tables are created and data generated for each table.

Once the conventional database is ready, the user is asked to initiate the creation of a Reduced database (the second storage Strategy discussed earlier). The data already generated for the conventional database is used to create and populate the three call tables in this Strategy.

The Reduced, Atomized Database (the third Strategy), too, is generated in like manner from the conventionally stored data.

### 5.3 Implementation of Schema Algebra operators

Six operators from each of the three Strategies under consideration were taken up for detailed coding and experimental study: three “traditional” relational algebra operators and three operators specific only to SA. They are:

1. Selection
2. Projection
3. Join
4. Querying with Single Pattern (this will also be referred to later in an abbreviated form as gamsing)
5. Querying with Multiple Pattern (sometimes abbreviated to gamult)
6. Creating and populating relations with schemas (abbreviated to var-rho)

Of these, the first five are merely querying operators which do not in any way modify the database, whereas the sixth is a restructuring operator which can create new tables, add new columns to existing tables and also insert rows (full or partial) to the tables.

The implementation interface allows the user to choose any operation under any strategy and specify the parameters for the operation, such as the table(s) to be used,
the selection or join condition to be applied, etc. Once all parameters (if required) for the operation are filled in, the user initiates the operation and the program displays the time taken (in milliseconds) for the operation to complete.

### 5.3.1 Experimental Results and Graphs

We now present the actual results of some experiments conducted on the test bed described earlier. Execution time for the various operators were recorded under varying conditions. As has been already mentioned, \( N_t \) refers to the number of rows in the table, \( J_d \) to the Join Density and \( S_d \) to the Selection Density.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>934</td>
<td>1263</td>
<td>2307</td>
</tr>
<tr>
<td>Projection</td>
<td>879</td>
<td>1043</td>
<td>1318</td>
</tr>
<tr>
<td>Join</td>
<td>1043</td>
<td>10239</td>
<td>13193</td>
</tr>
<tr>
<td>Gamsing</td>
<td>1813</td>
<td>1263</td>
<td>2142</td>
</tr>
<tr>
<td>Gamult</td>
<td>2142</td>
<td>3186</td>
<td>1648</td>
</tr>
<tr>
<td>Var-rho</td>
<td>16066</td>
<td>7307</td>
<td>14499</td>
</tr>
</tbody>
</table>

Table 5.1: Execution time with table size of 500, join density 0.25

Table 5.1 shows the results of running the six operations on two tables of 500 tuples and 3 columns each, with a selection density of 3 for the column referred to in the selection condition and a join density of 0.25 for the columns referred to in the join condition. \( S_1 \) refers to the "conventional" strategy, \( S_2 \) to the "reduced" strategy and \( S_3 \) to the "reduced, atomized" strategy. All times are given in milliseconds.

The tables which follow, ie, Tables 5.2 to 5.10 show the results when tuple numbers and join densities are varied.

After each table, we present two graphs to illustrate the time variations involved in the operations. The first graph, titled Individual Operations plots the six operations against the time taken, for each of the three storage strategies studied. The second graph is titled Mix of Operations, and the motivation behind offering this view is as follows. Processing a SchemaLog program typically consists of a mixture of various operations from \( S.A \). To simulate this scenario, we have manually combined
the results of our timing experiments in varying proportions. The mixes we have investigated are:

1. 100% querying operations
2. 75% querying and 25% restructuring operations
3. 50% querying and 50% restructuring operations
4. 25% querying and 75% restructuring operations
5. 100% restructuring operations

Within each of the above combinations, the load for querying is shared equally among the different querying operations of \(S.A\) and the load for restructuring is distributed likewise. Among the six operators we have studied, the first five are strictly querying operators and the last alone is a restructuring operator. Then, a mixture of 75% querying and 25% restructuring operations would mean that each of Selection, Projection, Join, Gamsing and Gamult would be allocated 15% of the work load and Var-Rho would be given 25%. It is our belief that such a study would better reveal the performances of the various Strategies than specifically chosen SchemaLog programs would.

**Graph Labels**

In all the graphs that follow:

- the values on the Y-axis denote the time, in milliseconds
- the values on Z-axis denote the three storage Strategies, viz:
  - \(S1\) = Conventional Strategy
  - \(S2\) = Reduced Strategy
  - \(S3\) = Reduced, Atomized Strategy

In graphs titled “Individual Operations”, the values on the X-axis refer to the six operators, viz:

- \(Op1\) = Selection
- \(Op2\) = Projection
- \(Op3\) = Join
\( Op4 \) = Gamsing \( (\text{Single Pattern Querying}) \)
\( Op5 \) = Gamult \( (\text{Multiple Pattern Querying}) \)
\( Op6 \) = Var-rho \( (\text{Creating relations, schemas; adding data}) \)

In graphs titled “Mix of Operations”, the values on the X-axis refer to the five different mixes of operations, viz;

M1: 100\% querying operations
M2: 75\% querying and 25\% restructuring operations
M3: 50\% querying and 50\% restructuring operations
M4: 25\% querying and 75\% restructuring operations
M5: 100\% restructuring operations

The two graphs for Table 5.1 are in figures 5.1 and 5.2.

![Graph](image)

Figure 5.1: Individual Operations (from Table 5.1)
Figure 5.2: Mix of Operations (from Table 5.1)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>934</td>
<td>1208</td>
<td>2142</td>
</tr>
<tr>
<td>Projection</td>
<td>989</td>
<td>1099</td>
<td>1648</td>
</tr>
<tr>
<td>Join</td>
<td>1154</td>
<td>19528</td>
<td>26711</td>
</tr>
<tr>
<td>Gamsing</td>
<td>1812</td>
<td>1043</td>
<td>2307</td>
</tr>
<tr>
<td>Gamult</td>
<td>2527</td>
<td>3405</td>
<td>1648</td>
</tr>
<tr>
<td>Var-rho</td>
<td>29330</td>
<td>13021</td>
<td>27684</td>
</tr>
</tbody>
</table>

Table 5.2: Execution time with table size of 1000, join density 0.25
Figure 5.3: Individual Operations (from Table 5.2)

Figure 5.4: Mix of Operations (from Table 5.2)
Table 5.3: Execution time with table size of 2500, join density 0.25

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>934</td>
<td>1209</td>
<td>2417</td>
</tr>
<tr>
<td>Projection</td>
<td>1318</td>
<td>1373</td>
<td>3594</td>
</tr>
<tr>
<td>Join</td>
<td>1538</td>
<td>60642</td>
<td>63741</td>
</tr>
<tr>
<td>Gamsing</td>
<td>2802</td>
<td>1098</td>
<td>2032</td>
</tr>
<tr>
<td>Gamult</td>
<td>3515</td>
<td>3845</td>
<td>1813</td>
</tr>
<tr>
<td>Var-rho</td>
<td>67887</td>
<td>40893</td>
<td>80395</td>
</tr>
</tbody>
</table>

Figure 5.5: Individual Operations (from Table 5.3)
Figure 5.6: Mix of Operations (from Table 5.3)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>934</td>
<td>1374</td>
<td>2197</td>
</tr>
<tr>
<td>Projection</td>
<td>2417</td>
<td>1483</td>
<td>6956</td>
</tr>
<tr>
<td>Join</td>
<td>1978</td>
<td>111752</td>
<td>115703</td>
</tr>
<tr>
<td>Gamsing</td>
<td>4339</td>
<td>1153</td>
<td>1978</td>
</tr>
<tr>
<td>Gamult</td>
<td>5273</td>
<td>3570</td>
<td>1757</td>
</tr>
<tr>
<td>Var-rho</td>
<td>120767</td>
<td>69208</td>
<td>164790</td>
</tr>
</tbody>
</table>

Table 5.4: Execution time with table size of 5000, join density 0.25
Figure 5.7: Individual Operations (from Table 5.4)

Figure 5.8: Mix of Operations (from Table 5.4)
\[ N_t = 10000; J_d = 0.25; S_d = 3 \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>1977</td>
<td>1977</td>
<td>2526</td>
</tr>
<tr>
<td>Projection</td>
<td>6316</td>
<td>3515</td>
<td>25114</td>
</tr>
<tr>
<td>Join</td>
<td>7634</td>
<td>249817</td>
<td>236131</td>
</tr>
<tr>
<td>Gamsing</td>
<td>7634</td>
<td>1208</td>
<td>2033</td>
</tr>
<tr>
<td>Gamult</td>
<td>8238</td>
<td>4064</td>
<td>2142</td>
</tr>
<tr>
<td>Var-rho</td>
<td>271672</td>
<td>182963</td>
<td>411960</td>
</tr>
</tbody>
</table>

Table 5.5: Execution time with table size of 10000, join density 0.25

Figure 5.9: Individual Operations (from Table 5.5)
Figure 5.10: Mix of Operations (from Table 5.5)
\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
$N_t = 500; J_d = 0.5; S_d = 3$ & \\
Operation & Time(S1) & Time(S2) & Time(S3) \\
\hline
Selection & 933 & 1208 & 2252 \\
Projection & 879 & 1154 & 1373 \\
Join & 1044 & 18222 & 25680 \\
Gamsing & 1647 & 1098 & 1977 \\
Gamult & 2362 & 3185 & 1868 \\
Var-rho & 15626 & 7469 & 14172 \\
\hline
\end{tabular}
\caption{Execution time with table size of 500, join density 0.5}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
$N_t = 1000; J_d = 0.5; S_d = 3$ & \\
Operation & Time(S1) & Time(S2) & Time(S3) \\
\hline
Selection & 989 & 1373 & 2526 \\
Projection & 1098 & 1209 & 1758 \\
Join & 1153 & 38117 & 50258 \\
Gamsing & 2252 & 1098 & 2362 \\
Gamult & 2856 & 3295 & 1867 \\
Var-rho & 27464 & 12906 & 28338 \\
\hline
\end{tabular}
\caption{Execution time with table size of 1000, join density 0.5}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
$N_t = 2500; J_d = 0.5; S_d = 3$ & \\
Operation & Time(S1) & Time(S2) & Time(S3) \\
\hline
Selection & 879 & 1373 & 2527 \\
Projection & 1373 & 1538 & 3805 \\
Join & 1868 & 110357 & 118990 \\
Gamsing & 3240 & 1099 & 2307 \\
Gamult & 3954 & 3680 & 2087 \\
Var-rho & 67035 & 29504 & 79095 \\
\hline
\end{tabular}
\caption{Execution time with table size of 2500, join density 0.5}
\end{table}
Figure 5.11: Individual Operations (from Table 5.6)

Figure 5.12: Mix of Operations (from Table 5.6)
Figure 5.13: Individual Operations (from Table 5.7)

Figure 5.14: Mix of Operations (from Table 5.7)
Figure 5.15: Individual Operations (from Table 5.8)

Figure 5.16: Mix of Operations (from Table 5.8)
\[ N_t = 5000; J_d = 0.5; S_d = 3 \]

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>988</td>
<td>1483</td>
<td>2362</td>
</tr>
<tr>
<td>Projection</td>
<td>1813</td>
<td>1702</td>
<td>6840</td>
</tr>
<tr>
<td>Join</td>
<td>2033</td>
<td>229932</td>
<td>234200</td>
</tr>
<tr>
<td>Gamsing</td>
<td>5053</td>
<td>1044</td>
<td>2472</td>
</tr>
<tr>
<td>Gamult</td>
<td>5657</td>
<td>4099</td>
<td>2087</td>
</tr>
<tr>
<td>Var-rho</td>
<td>125956</td>
<td>73993</td>
<td>173010</td>
</tr>
</tbody>
</table>

Table 5.9: Execution time with table size of 5000, join density 0.5

Figure 5.17: Individual Operations (from Table 5.9)
Figure 5.18: Mix of Operations (from Table 5.9)

<table>
<thead>
<tr>
<th>Operation</th>
<th>Time(S1)</th>
<th>Time(S2)</th>
<th>Time(S3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Selection</td>
<td>988</td>
<td>1923</td>
<td>2362</td>
</tr>
<tr>
<td>Projection</td>
<td>6811</td>
<td>3240</td>
<td>25749</td>
</tr>
<tr>
<td>Join</td>
<td>8624</td>
<td>471871</td>
<td>494273</td>
</tr>
<tr>
<td>Gamsing</td>
<td>8184</td>
<td>1154</td>
<td>2581</td>
</tr>
<tr>
<td>Gamult</td>
<td>8513</td>
<td>4339</td>
<td>1923</td>
</tr>
<tr>
<td>Var-rho</td>
<td>262709</td>
<td>212662</td>
<td>418500</td>
</tr>
</tbody>
</table>

Table 5.10: Execution time with table size of 10000, join density 0.5
Figure 5.19: Individual Operations (from Table 5.10)

Figure 5.20: Mix of Operations (from Table 5.10)
5.4 Analysis of Results

5.4.1 Individual Operations

The various graphs where the performance of the various operators is plotted against time (under the three Strategies) show great similarities. Refer to graphs in Figures 5.1, 5.3, 5.5, 5.7, 5.9, 5.11, 5.13, 5.15, 5.17 and 5.19. It is quite clear that for the purely querying operations (operators 1 to 5 in the graphs), the Conventional storage strategy is decidedly superior. The other two Strategies show comparable performances for all the querying operators but the Join. For the Reduced Strategy, the peak in the graph for the Join can be attributed to the fact that the join is made only on a fragment of the original conventional tuple and the full tuples have to be reassembled once the results of the join are known.

To illustrate, consider the data in the conventional tables given in Figure 4.1. The first table, person contains the tuple (John,35,Montreal) and the second, works.in, contains the tuple (John,CIBC). In the Reduced form these two “conventional” tuples would produce the following five tuples:

\[
\begin{align*}
<person,t_1,name,John > \\
<person,t_1,age,35 > \\
<person,t_1,city,Montreal > \\
<works.in,t_2,name,John > \\
<works.in,t_2,company,CIBC >
\end{align*}
\]

Now if a join of person and works.on on the column name were required, the “conventional” join would produce the single tuple <John,35,Montreal,CIBC>. In the Reduced strategy, the joinability of \( t_1 \) and \( t_2 \) would have to be determined by looking for a match on name (here, \(<person,t_1,name,John > \) and \(<works.in,t_2,name,John > \)). A new tuple id (as well as a new table name) has then to be assigned to all tuples with the existing id’s of \( t_1 \) and \( t_2 \) (to indicate that they can be reconstituted into just one tuple in “conventional” form). In our case, the reduced tuples after the join would be:

\[
\begin{align*}
<newtable,t_3,name,John > \\
<newtable,t_3,age,35 >
\end{align*}
\]
A similar explanation can be given for the peak in the join plot for the Reduced, atomized storage. Here, too, the joinable tuples are determined, their tid's established, and the joined tuples assembled using the tid.

As regards restructuring operations (operator 6 in the graphs), the cost in the Conventional Strategy is expectedly high. The operation involves such costly elements as changing table schema, creating new tables, etc. And again, as expected, the Reduced Strategy has a low cost for restructuring. There are no new tables created, nor is the existing schema changed. New tuples are merely added to existing tables. The peak for restructuring in the plot for the Reduced, atomized storage may be surprising, at first sight. True, some new tables may need to be created when attributes are added to existing schema, but on the whole here the operation involves adding rows to existing tables. On closer examination of the practical effects of the restructuring, we, however, see that the various tables being created/modified have to be successively opened and closed as tuples from the input "schemaless" relation are being processed. Opening/closing of data sets is an operation that does incur some overhead. This cost is essentially responsible for the observed peak for this Strategy.

5.4.2 Mix of Operations

Figures 5.2, 5.4, 5.6, 5.8, 5.10, 5.12, 5.14, 5.16, 5.18 and 5.20 show the cost of a program that has a mix of operations (as explained earlier in this chapter). We see that in an application used predominantly for querying operations, the conventional storage strategy is superior. Querying costs for the reduced storage strategy are somewhat higher. But as we move towards a greater mix of restructuring operations, we see that the cost for conventional storage strategy (as also for the Reduced, atomized strategy) keeps on increasing sharply whereas the cost for the reduced strategy remains fairly constant. In a SchemaLog application that has typically more restructuring operations than querying operations, the preferred strategy should then be the Reduced strategy. A mixture of the two strategies, in which the Conventional storage is used for base relations and the Reduced strategy for derived relations would appear to be appropriate. In this context it is worth noting that many emerging database applications such as “Online Analytical Processing” (OLAP) technology ([CCS95])
have to deal with a considerable amount of restructuring operations.

5.4.3 **Effect of table size on cost of SA operations**

Yet another interesting use of the time data in the experimental tables given earlier is to study the cost of the individual operations (in the various Strategies) with regard to the size of the database tables. In our experiments, the size of the tables was varied from 500 tuples to 10,000 tuples. There follow now the graphs corresponding to the six operators studied.

**Graph Labels**

The labels on the Y-axis refer to time in milliseconds.

The labels on the X-axis refer to the size of a table

1 = 500 tuples
2 = 1000 tuples
3 = 2500 tuples
4 = 5000 tuples
5 = 10,000 tuples

S1 refers to the *Conventional* Strategy
S2 refers to the *Reduced* Strategy
S3 refers to the *Reduced, Atomized* Strategy
Figure 5.21: Selection

Figure 5.22: Projection
Figure 5.23: Join

Figure 5.24: Single Pattern Querying (gamsing)
Figure 5.25: Multiple Pattern Querying (gamult)

Figure 5.26: Creating Tables, Schema; Adding Rows (Var-Rho)
Some points worth noting after a study of the graphs are:

1. In the Conventional Strategy, the cost of Selection and Join remain fairly constant, irrespective of the size of the tables. This can be explained by the fact that these operations work with only the "selected" and the "joinable" tuples which are directly available from indexes, and exhaustive reads through the tables are not required.

2. In the other two strategies, selection costs are fairly constant, but at a considerably higher level than for the Conventional strategy. But Join costs rise quite dramatically as the table size is increased. For any considerably-sized database a join in the two non-conventional strategies is an expensive operation.

3. Restructuring costs rise with increase in table size, in all three strategies. But the rise is least steep in the Reduced Strategy.

5.5 Conclusion

This Chapter presented the results of the experiments we conducted in implementing S.A operators under the three storage strategies in Chapter 4. The results comparing the performances were tabulated in detail for varying conditions such as table size, join density and selection density. Performances of the various strategies for a mix of different operations (as likely to occur in a real-life scenario) were also illustrated graphically. Another angle studied was the comparative performance of various operations when table size was varied. The chapter concluded with a summary the findings of the experiments. The most significant of these is that in conventional querying operations, the "conventional" storage strategy is decidedly superior and must be retained as the strategy of choice. But when restructuring operations are involved, the "reduced" strategy has a clear edge over the others. And since modern applications tend to have an increasing proportion of restructuring operations, this strategy must find a place in an efficiency-oriented implementation.
Chapter 6

Conclusion

In this Chapter, we compare our work to some similar ones done in the field. We then summarise the principal contributions of this thesis and conclude with mentioning the possibilities for future work in the area.

6.1 Comparison

Here, we shall compare our work with similar work related to the implementation of other higher-order logic database languages. We specifically consider four such implementations:

1. Implementation of F-logic [KLV95], one of the most comprehensive logical accounts for the object-oriented data model.

2. Implementation of Golog, an object-oriented logic developed at Griffith University, Australia [Dob95].

3. Implementation of HiLog, a higher-order database logic programming language developed at SUNY, Stony Brook [CKW89].

4. Implementation of F-logic in FLORID, developed at the University of Freiburg, Germany.

Our comparison will deal with the implementations rather than the language features. For a comparison of the languages, interested readers are referred to [LSS96].
In [Law93], Lawley describes the implementation of an interpreter for F-logic. This is achieved by translating the F-logic syntax into an appropriate representation in NU-Prolog, Quintus, Eclipse or XSB Prolog. As described by the author, the goal of this implementation was simplicity — indeed the interpreter is just a few lines of meta-programming code — rather than efficiency. While this is useful for running small F-logic programs, it is not clear how this can be used in a real database context.

In [Lef93] Lefebvre describes an implementation of Golog, as a declarative query language for a deductive object-oriented database, with F-logic acting as an “application programming language”. This implementation bootstraps on the implementation of the F-logic interpreter above and inherits its limitations.

Sagonas and Warren [SW95] describe an efficient implementation of HiLog within the WAM (Warren Abstract Machine) framework. Their idea is based on using a first-order translation of HiLog different from the one used for the proof of first-order semantics by the authors of HiLog. HiLog runs on top of XSB Prolog and fully exploits the run time optimization of XSB Prolog. HiLog can, in fact, be implemented thus in any Prolog system simply by changing its input/output predicates to support terms that are expressed using higher-order syntax. The authors show experimentally that Hilog programs that do not use any higher-order features execute at the same speed as Prolog programs and that generic Hilog predicates, when compiled using their special compilation scheme, execute at least an order of magnitude faster than generic Prolog predicates.

This implementation again, though efficient, runs a HiLog program ultimately as a Prolog program. The authors, too, view their work as a compile time program specialisation pre-processing step. There is no attempt at a straight-forward translation of the higher-order Hilog syntax into a correspondingly expressive procedural language.

FLORID, [FHKS97], developed at the Universities of Mannheim and Freiburg, is described by its authors as a prototype environment in which the practical programming aspects of F-logic can be tested. It is claimed that, in contrast to the Golog implementation, nearly all the distinctive features of F-logic have been realized in FLORID. The prototype supports such aspects of the object-oriented model as multiple, non-monotonic inheritance. The evaluation strategy is bottom-up, using an extension of Datalog methods. The authors themselves state that efficiency was not
the goal of this implementation, but rather the demonstration of all the features of
the language of F-logic.

We remark here that, while useful as a testing medium for the extensive features
of the F-logic language, FLORID does not seem, in its present form, to be suitable
in the area of large databases.

In contrast with all the above implementations, our implementation of SchemaLog
has the following unique features:

1. It is not based on translation into any other language like Prolog or encoding
into a lower-order syntax. Rather, our implementation is direct and follows a
straightforward translation of SchemaLog rules into a corresponding sequence
of procedural SA operations which, we have shown, can be implemented under
various physical storage architectures.

2. Schema Algebra is at the core of our implementation. This is especially suited
for set-oriented processing which is more appropriate for a database context as
opposed to a purely logic programming context.

3. To our knowledge, issues like meta-data querying and piecemeal computation
have not been dealt with in previous implementations.

4. One of our main objectives was efficiency of implementation in a large database
context. To this end, we proposed alternative strategies for storage and handling
of data, and evaluated their effectiveness with a series of experiments.

6.2 Summary

This thesis forms part of an on-going project on the full implementation of SchemaLog, an
advanced database programming language. SchemaLog, with its higher order logical
syntax, has the capability of performing a variety of tasks that are essential in the
current state of information technology. We therefore began this dissertation with an
introduction to the basic theory of SchemaLog, its syntax and semantics. We followed
it up with detailed definitions and illustrations of the operators of Schema Algebra.
Since our implementation of SchemaLog was restricted to the single database context,
relevant modifications were made to the original definitions of $\text{SchemaLog}$ syntax and of $\text{SA}$ operators. It is upon these $\text{SA}$ operators that our implementations are based.

The rules in a $\text{SchemaLog}$ program can, like all logic language programs, be evaluated top-down or bottom-up. We have chosen a top-down implementation. And since we are working with databases expecting a set of tuples as answers to queries, we chose the Rule/Goal Tree evaluation Method for the $\text{SchemaLog}$ rules. We presented a complete set of algorithms to deal with the RGT evaluation process as applied to $\text{SchemaLog}$ programs.

Realising that some of $\text{SchemaLog}$'s novel constructs, which led to the definition of some novel operators in the extended relational algebra called $\text{SA}$, would need some novel methods of physical storage of data, we proposed three different storage strategies. One of these is an extension of the conventional method, but the other two were devised with particular attention to $\text{SchemaLog}$ requirements such as restructuring and piecemeal computation of tuples. We described the three strategies and presented detailed algorithms for implementing the $\text{SA}$ operators in all strategies, as well as theoretical cost estimates for the implementation.

We followed this up with the presentation of the results of some extensive experimental tests conducted on the $\text{SA}$ operators under the various strategies. From practical considerations and from the results of the experiments, we can recommend that the most efficient way to implement $\text{SchemaLog}$ programs would be to use the “conventional” storage for existing database relations, and the “reduced” strategy for the derived database relations.

### 6.3 Future Work

As we have stated on several occasions during the course of this thesis, this implementation of $\text{SchemaLog}$ has been restricted to the single database context. Hence much work remains to be done in order to achieve a full implementation of the many aspects of this powerful programming language. An implementation of $\text{SchemaLog}$ for multi-database interoperability among a federation of INGRES databases has been done and is described in [LSPS95].

As noted in [Sub97], $\text{SA}$ is not a sufficiently powerful language to express every program in the full-fledged $\text{SchemaLog}$ language. As it stands, not all $\text{SchemaLog}$
programs can be translated into an equivalent $SA$ expression. When $SA$ has been sufficiently developed to this purpose, work needs to be done on an implementation that converts any given SchemaLog program to an equivalent $SA$ expression and sends it on for evaluation. In our work, we have supplied algorithms for a top-down implementation and recommended physical storage structures that can efficiently hold permanent and temporary tables during evaluation. There is as yet no single implemented system that is able to achieve all of the above at the same time.

We have proposed a top-down method of evaluating SchemaLog programs by adapting the classical RGT approach to the special needs of SchemaLog. Logic programs can also be efficiently evaluated bottom-up. Research into how the bottom-up evaluation methods of “Semi-Naive” coupled with “Magic Sets” can be tailored to the requirements of SchemaLog can provide an interesting alternative to evaluating SchemaLog programs.

Implementations of other aspects of SchemaLog and of SchemaSQL – a systematic extension of standard SQL with the capabilities of SchemaLog features – are already in progress. We feel confident that the contributions of this thesis will form part of an integrated system that could fully realize in practice the extensive theoretical possibilities thrown up by SchemaLog.
Bibliography


