nD-SQL: EXTENDING SCHEMASQL TOWARDS MULTIDIMENSIONAL DATABASES AND OLAP

Frédéric Gingras

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 1998

0-612-39112-4

CONCORDIA UNIVERSITY
School of Graduate Studies

This is to certify that the thesis prepared

By: **Frederic Gingras**

Entitled: ***n*D-SQL: Extending SchemaSQL Towards Multidimensional Databases and OLAP**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_______________ Chair

_______________ Examiner

_______________ Examiner

_______________ Supervisor

Approved _______________
Chair of Department or Graduate Program Director

_______________ 19 ____ _______________

Dr. Nabil Esmail, Dean
Faculty of Engineering and Computer Science

Abstract

nD-SQL: Extending SchemaSQL Towards Multidimensional Databases and OLAP

Frédéric Gingras

Decision support systems are becoming tools of basic necessity for corporations big and small, but the costs in time and money to integrate the various sources containing the data to be analysed is often prohibitive. In view of those costs, a *non-intrusive* solution would be an attractive alternative to porting data and applications from legacy systems to a common platform. In this context, one of the key problems for interoperability lies in the heterogeneity in schema of the underlying sources. We propose a combined solution to the above problems. This thesis proposes a formal model for a federation of relational databases with possibly heterogeneous schemas. The Federation Model is comprehensive enough for: (i) capturing the diversity of schemas arising in practice, allowing a symmetric treatment of data and schema, and (ii) capturing the complete space of dimensional representations of data, fully exploiting the n logical dimensions structured along the three physical dimensions implicit in the relational model – row, column, and relation. An n-dimensional query language called nD-SQL is also proposed. This language makes use of the Federation Model and is capable of: (a) resolving schematic discrepancies among a collection of relational databases or data marts with heterogeneous schemas, and (b) supporting a whole range of multiple granularity aggregation queries like CUBE, ROLLUP, and DRILLDOWN, but, to an arbitrary, user controlled, level of resolution. In addition, nD-SQL can express queries that restructure data conforming to any particular dimensional representation into any other. The semantics of nD-SQL is downward compatible with the popular SQL language. The thesis also proposes an extension to relational algebra, capable of restructuring, called *restructuring relational algebra* (RRA). We use RRA as a vehicle for efficient processing of nD-SQL queries, and we propose an architecture for this purpose. We develop query optimisation strategies based on the properties of RRA operators. We have implemented the operators of the RRA and we have tested the performance of heuristics developed for query optimisation.

To my mother, my father and Annie.

Each loving and helping me in their own, special way.

Acknowledgments

So many people have shaped who I am today, in small or enormous measures, that properly acknowledging them all would take more pages than the length of this thesis. I shall nonetheless endeavour to thank here as many as I can.

I would like to thank my mother and my father, who have always encouraged me to be true to myself. They believe in me and have taught me to do the same. I could never thank them enough for the support, love and friendship they have given me over the last twenty-five years.

I would also like to thank Annie, my love. You have helped me stay sane (that is as sane as I was before I met you), reaching out to me when I needed it most, bringing a smile to my face when my troubles seemed overwhelming. Your love sustains and strengthens me and I hope that my own love can do the same for you.

To all the friends who have endured me over the years, I say thank you (and secretly wonder how you managed it). All of you, fans of science-fiction or fantasy, role-players, card players, pool players, bowling players or badminton players. Dilletantes in movie soundtracks, in theoretical, particle or astro-physics. Critiques who studied and debated with me the fine points of the impact theory of mass extinctions. All those who shared with me their appreciation of Loreena McKennitt or Sarah McLachlan. I must thank you all from the bottom of my heart for your friendship, your help, your time and your positive influence on my life. If a person could be the sum of who his friends are, I know I would be a much better person than I can ever hope to become.

Over the years, the numerous teachers who taught me have managed to kindle my interest in each of their subjects of predilection. They also recognised my interests and abilities and slowly guided me on this marvelous journey of discoveries. I would like to thank them all for the incredible opportunity they gave me in teaching me to look at the world in a myriad of different ways.

Special thanks must go to my thesis advisor, Dr. Laks Lakshmanan. Laks' knack for taking the best parts of what you said, and massaging those into ideas that can be worked upon, never ceases to amaze me. His interests are so varied and his mind so agile that it is always a pleasure to converse with him on any subject. Even when we do not agree on something, there is always room for discussion and in the end, a better understanding emerges. Laks, if it was in my power to do so, I would find you a whole department of students to work on your projects. I know that you would have something fascinating for each of them to do and that in the end, the next generation of database technology would become a reality very fast.

I would also like to gratefully acknowledge the financial support I received from the J.W. McConnell Memorial Fund, and the NSERC grants I obtained through Laks. I also made use of these awards for attending conferences.

Last, but not least, I would like to thank the members of my extended family, blood related or otherwise. Your interest in me, kindness and warmth keeps on helping me along the path of success.

Someone wise once said that when you travel, what should matter is not the destination but the journey itself. The Chinese people also have a saying, which can be a blessing as well as a curse: "May you live in interesting times". Well, life's journey has been incredibly interesting up to now, and I certainly do not want to reach the destination anytime soon. May I be so lucky as to keep meeting people as extraordinary as all of you who touched my life up to now. Thank you again everyone. This is dedicated to all of you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

We are rapidly reaching the end of the twentieth century and in the span of a few decades, our needs in terms of information processing and analysis have reached a point where powerful automated systems are no longer a nice thing to have, but a basic necessity. The database community has thus been forced to adopt new paradigms centered on the concepts of efficiency, productivity and competitivity, paradigms like data warehouses, on line analytical processing (OLAP) and data mining. These paradigms are all proposed as components of decision support systems.

One of the most important facts that has to be dealt with in this context is *interoperability* : if it can be helped, users do not want to throw away their existing systems. Replacing them would involve costs in time and money that must be avoided. But empowering existing systems with the new functionalities needed, and integrating these systems in a greater whole that would enable seamless cross-querying, is one of the biggest challenges faced by today's database community.

The interoperability problem entails the resolution of incompatibilities and conflicts on a number of different fronts, including: the myriad of platforms in use today; the differences in schema of databases containing information of similar nature; the variety of transaction management systems; the syntactic differences between languages used to access data in each system; and numerous others.

Some of these problems arise because the various sources of data, often inside a single organisation, have been designed by different people, at different points in time, to cater to different needs, and in most cases to be autonomous. The same autonomy that was a blessing when it enabled distinct entities to administrate, query, update and restructure data, has quickly become a curse of a sort. The tradeoff between autonomy and the need for integration appears clearer today than ever before in corporations that need powerful, integrated decision support tools. For example, the Canadian telecommunication company Bell Canada has created over the years numerous systems storing information about different aspects of its network of cables spread over an immense territory. Today, in order to make sound business decisions about the assignment of network repair crews and maintenance and installation budgets, the corporation painstakingly developed an in-house tool that permits it to incorporate data from its various systems. This tool was designed to analyse in various ways the data coming from all the different sources, without necessitating any changes in the individual systems.

As this example tries to illustrate, there is a real need for a sound and formal basis to facilitate interoperability between different database systems. It would have made life much easier for Bell Canada if there was a system capable of integrating those components DBMS and interoperate among them. Such systems should not have to be built using ad hoc designs, particular to each individual case.

It has been proposed in the literature that data-mining and OLAP being computationally expensive, they are best supported by a data warehouse. But, as discussed in [CD97], building a data warehouse is a long, complex, and costly process, often taking up to several years to complete. Many organisations adopt an intermediate solution, whereby they create the so-called data marts, which are essentially miniature data warehouses integrating small subsets of the operational databases. At the opposite end of the spectrum, some organisations tend to adopt a *virtual warehouse* approach, at least for a limited time, before they analyse their needs and customise their systems. *Thus, in the evolutionary life-cycle of a data warehouse, one has to cope with interoperating among operational databases, among data marts, and among both.* Given the ultimate need to perform OLAP-style computations, it would be desirable to have one query language that can express not only conventional queries across component databases (or data marts), but also OLAP-style queries.

It has been recognised that even in the apparently simple context of a federation consisting of relational databases, the conflict among the component schemas raise serious challenges for interoperability. For instance, an entry such as "ibm" might appear as a domain value in one component database, as an attribute in another, and as a relation name in the third (see Figure 1 for an example federation of relational databases). It is known that conventional languages like SQL or variants cannot be used to overcome this conflict (see [LSS96]), without a host language.

This thesis addresses the dual problem of solving the above kinds of interoperability conflicts between relational sources, while enabling OLAP-style computations to be performed on that data.

Ticker	Date	Measure	Price
ibm	10|27|97	open	63.67
ibm	10|27|97	close	62.56
...	...	...	...
ms	11|01|97	open	44.60
ms	11|01|97	close	46.17
...	...	...	...

(a)`nyse::prices`

Stocks	Date	open	close	...
ibm	10|27|97	63.67	62.56	...
...	...	...	...	...
ms	11|01|97	44.60	46.17	...

(b)`tse::quotes`

Date	open_ibm	open_ms	open_...	...	close_ibm	close_ms	close_...
10|27|97	63.67	50.23	...	...	62.56	48.54	...
...	...	...	...	...	...	...	...
11|01|97	65.23	44.60	...	...	63.05	46.17	...

(c) `bse::prices`

Date	open	close	...
10|27|97	63.67	62.56	...
...	...	...	...
11|01|97	65.23	63.05	...

`mse::ibm`

Date	open	close	...
10|27|97	50.23	48.54	...
...	...	...	...
11|01|97	44.60	46.17	...

`mse::ms`

...

(d) relations in `mse`

Figure 1: A federation of relational databases with heterogeneous schemes, containing stock market data.

Only relevant relations from each database are shown. The notation `db::rel` means `db` is a database containing relation `rel`. The same data is used in every database to clearly show the schema correspondence.

To properly contrast the thesis with the existing research, the next section reviews some of the existing works in the field.

1.1 Current State of Research

The problem of presenting to users an integrated view of the data in multiple database systems has been studied by researchers for a long time. Surveys on this subject are found in [ACM90], and [ACM94]. A more focused discussion can be found in [S97]. Another important reference on the subject is Won Kim ([KCGS93]). The main approaches are: (i) mapping component databases to a common canonical data model and (ii) using a non-procedural language for cross-querying.

The common data model approach consists in mapping the databases of a federation to a common model that is rich enough (in terms of modelling power) to capture their similarities in information content. Often, users will be presented a 'view' of the various sources that all fit the user's perception of his or her data. This way a user will not need to know that there exist any schematic discrepancies between the sources. Examples of the use of the common data model approach are: the Multibase project([LR9289]), the Mermaid project ([Tem87]), and the Pegasus project ([ASD+91]).

The non-procedural language-based approach has been very popular over the years. It consists in using a language that allows users to define and manipulate several autonomous databases in a non-procedural fashion. This approach has the advantage of giving more flexibility, since the common data model is not necessary and is, in some sense, redefined each time dynamically by the use of the language. Much work has been done following this approach. The following is a selection from those with which we compare our work in Chapter 7: [Lit89, ASD+91, KKS92, Bee93, CL93, GLRS93, SSR94, KGK+95, MR95, Cat96, SQL96].

More recently, a family of related works has been proposed using higher order languages as vehicles for information integration. We note *SchemaLog*([Andr+96]), a logic language, Tabular Algebra ([GLS96]), and the SchemaSQL language ([LSS96]), a SQL-compatible language which was the starting point for our work. A comparison of our work with SchemaSQL can also be found in Chapter 7.

Industry efforts to solve the interoperability problem have also made their way into commercial products. Among those, we note Oracle/SQL (from Oracle Corp.) ([ORA]) and Data Joiner, from IBM ([DB296]). The latter lets users combine relations coming from different sources, and query them as if they were in one (DB2) source. The work is promoted as a decision support tool.

1.2 Structure and Contributions of This Thesis

As stated earlier, even in the context of a federation consisting of relational databases, the conflicts among the component database schemas raise serious challenges for interoperability.

The body of work pertaining to the interoperability problem is impressive. However, none of the proposed common data models *or* non-procedural languages satisfactorily solves the problem. For each of the works mentioned in the preceding section, either: (i) there is an inherent rigidity present in the common data model proposed, which means costly changes to a system whenever an additional source must be integrated, or when changes are made to the schema of one of the sources; (ii) the language or model proposed is not rich enough to effectively solve the schema discrepancy problem; (iii) the language is not clearly downward compatible with SQL, which is the language of choice for most commercial DBMS in use today; (iv) the approach is intrusive insofar as it demands (sometimes extensive) changes to existing systems; or (v) the approach does not provide the necessary functionalities to incorporate OLAP-style computations.

Both of the main approaches have merit, but we contend that a hybrid approach can make use of the best of both worlds to cleanly resolve the problem. We believe that the need for mappings to a common schema calls for a high level query language capable of resolving schema conflicts automatically, assuming additional information on the component schemas is added to the federation in a *non-intrusive* manner.

Moreover, given the ultimate need to perform OLAP-style computations, this query language should not only be able to express conventional queries across component databases (or data marts), but also OLAP-style queries.

This thesis proposes a formal model for a federation of relational databases with possibly heterogeneous schemas. The model is comprehensive enough for: (i) capturing the diversity of schemas arising in practice, allowing a symmetric treatment of data and schema, and (ii) capturing the complete space of dimensional representations of data, which fully exploits the three physical dimensions implicit in the relational model – row, column, and relation. This separation of data dimensions from the physical dimensions will enable us to both solve the schematic heterogeneity problem for interoperability and to support OLAP-style computations. The Federation Model is presented in Chapter 2.

An n-dimensional query language called nD-SQL is then proposed. This language makes use of the Federation Model and is thus capable of: (a) resolving schematic discrepancies among a collection of relational databases or data marts with heterogeneous schemas, and (b) supporting a whole range of multiple granularity aggregation queries like CUBE, ROLLUP, and DRILLDOWN, but, to an arbitrary, user controlled, level of resolution. In addition, nD-SQL can express queries that restructure data conforming to any particular dimensional representation into any other. The semantics of nD-SQL is downward compatible with the popular SQL language. The syntax of nD-SQL is presented in Chapter 3 while its semantics is presented in Chapter 4.

The thesis also proposes an extension to relational algebra, capable of restructuring, called *restructuring relational algebra* (RRA). We use RRA as a vehicle for efficient processing of nD-SQL queries, and we propose an architecture for this purpose. We develop query optimisation strategies based on the properties of RRA operators. We have implemented the operators of the RRA and we have tested the performance of heuristics developed for query optimisation. Chapter 5 presents the RRA and discusses query processing and optimisation, while Chapter 6 presents and discusses the performance results for query optimisation. Then, nD-SQL and its approach are compared with related work in Chapter 7. Finally, Chapter 8 summarises and discusses our future work.

Chapter 2

The Federation Model

This chapter proposes a formal model for collections of relational databases that we call the Federation Model. The highlights of this model are: (i) It captures heterogeneous schemas of relational databases arising in practice, including cases where domain values in one database may appear as schema components in another; (ii) It gives a first class status to the three *physical dimensions* implicit in the traditional relational model – row, column, and relation; (iii) Using this, it gives a precise meaning to representations of n-dimensional data using three physical dimensions; (iv) it is straightforward to incorporate (relational) data marts with the federation model, and this is discussed at the end of the chapter.

2.1 The Federation Model: A Formal Model for a Federation of Relational Databases

Let us begin with the notion of a scheme. The size of practical database schemas may not be fixed and may be data dependent (e.g., the number of columns of `tse` and the number of relations in `mse`, in Figure 1). This problem is solved by proposing a "federation scheme". This notion makes it possible to view the scheme of a relation, a database, or a federation, as a *fixed* entity independent of the contents in it, just as in the classical case. Let us assume pairwise disjoint, infinite, sets of names, $\mathcal{N}$, values, $\mathcal{V}$, and id's, $\mathcal{O}$. Typewriter font is used for names (e.g., `Measure`) and Roman for values (e.g., open), regardless of what positions they appear in— data or relation/column label positions. Ids will always be clear from the context. The partial function $dom : \mathcal{N} \rightsquigarrow 2^{\mathcal{V}}$ maps names in $\mathcal{N}$ to their underlying domains of values. Names that only correspond to relations or databases do not have associated domains.

Definition 2.1.1 (Federation Scheme) *A federated name is a pair (N, X) where $N \in \mathcal{N}$ is a name and $X \subset \mathcal{N}$ is a finite subset of names, such that $N \notin X$. In a federated name, the component N is referred to as the* concept *and the set X as the associated* criteria set. *A federated name (N, X) is* simple *(resp.,* complex*) provided $X = \emptyset$ (resp., $X \neq \emptyset$). Simple federated names $(N, \emptyset)$ are usually denoted just as N, following the classical convention. A* federated attribute *or* relation name *is any federated name. A* federated relation scheme *is of the form $R(C_1, \ldots, C_n)$, where R is a federated relation name and the C_is are all federated attribute names. A* federated database scheme *is a set of federated relation schemes, and a* federation scheme *is a set of named federated database schemes.*

The intuition behind the above definition is two-fold: Firstly, a complex attribute name translates to a *set* of complex column labels in an instance. Similarly, complex relation names translate to a *set* of complex relation labels. For example, the complex attribute name (`Price`, {`Measure`, `Ticker`}) in the scheme might correspond in an instance to the set {`Price FOR Measure` = *low* AND `Ticker` = *ibm*, ..., `Price FOR Measure` = *close* AND `Ticker` = *hp*} of column labels. For example, the federation scheme of the instance shown in Figure 1 is:

$$\mathcal{S}_1 = \{\text{nyse} :: \text{prices(Ticker, Date, Measure, Price)}, \text{tse} :: \text{quotes(Ticker, Date, (Price, \{Measure\}))},$$
$$\text{bse} :: \text{prices(Date, (Price, \{Measure, Ticker\}))}, \text{mse} :: \text{(prices, \{Ticker\})(Date, (Price, \{Measure\}))}\}.$$

We use the notation *db::rel* to indicate that a relation *rel* is from database *db*. Notice that in the instance shown in Figure 1, the somewhat cryptic labels like "open" take the place of the formal label "`Price FOR Measure` = open". It will be shown later that the exact labels used are unimportant, and we will provide a clean mechanism for keeping track of their meaning.

Secondly, notice that the notion of a federated relation scheme formalises the idea that certain attribute domains are arranged along each of the three dimensions – relation, column, and row. Specifically, in an instance of a federated relation scheme (e.g., `mse::(prices, {Ticker})`(Date, (Price, {Measure}))), domain values of relation criteria (here `Ticker`) are placed along the relation dimension, domain values of criteria of complex columns (here `Measure`) are placed along the row dimension, and domain values of simple columns (here `Date`) are placed along the column dimension.

Definition 2.1.2 (Federation Instance) *Let $\mathcal{S} = \{\mathrm{d}_1 :: R_1(C_1, \ldots, C_k), \ldots, \mathrm{d}_m :: R_m(D_1, \ldots, D_n)\}$, the d_i not necessarily distinct, be a federation scheme. Then a federation instance (instance for short) of this scheme is a 7-tuple $\mathcal{I} = \langle \mathcal{D}$, rel, col, tup, conc, crit, val$\rangle$, defined as follows.*

- *$\mathcal{D} = \{d_1, \ldots, d_m\}$, i.e. $\mathcal{D}$ consists exactly of the distinct database names mentioned in the scheme $\mathcal{S}$.*
- *rel : $\mathcal{D} \to 2^{\mathcal{O}}$ is a function that maps each database name in $\mathcal{D}$ to a finite set of relation id's. Below, $\mathcal{R} = \bigcup_{d \in \mathcal{D}} rel(d)$ is used to denote the set of all relation id's in the instance.*
- *col : $\mathcal{R} \to 2^{\mathcal{O}}$ is a function that maps each relation id to a finite set of column id's.*
- *tup is a function that maps each relation id r in $\mathcal{R}$ to a finite set of tuples $tup(r)$ over the set of columns $col(r)$.*
- *conc : $\mathcal{O} \to \mathcal{N}$ is a function that maps each id to a name, called its underlying concept.*
- *crit : $\mathcal{O} \to 2^{\mathcal{N}}$ is a function that maps each id to a finite set of names, namely its underlying set of criteria.*
- *val : $\mathcal{O} \times \mathcal{N} \rightsquigarrow \mathcal{V}$ is a partial function that maps an id and a name (viewed as a possible criterion associated with the id) to a value.*

For example, an instance of the scheme $\mathcal{S}_1$ above is the federation shown in Figure 1, *intuitively speaking.* There are four database names— `nyse`, `tse`, `bse`, `mse`, each of them having their associated simple/complex relations. For instance, `mse` has the relations "ibm, ms, ...", each having the same set of column labels— "`Date`, low, high, ...". All these labels intuitively correspond to (relation and column) id's in the formal definition.

A small subset of the abstract instance corresponding to the federation of Figure 1 would be:
$\mathcal{D} = \{\texttt{nyse}, \texttt{tse}, \texttt{bse}, \texttt{mse}\}$
$rel(\texttt{nyse}) = \{\texttt{prices}\}$
$col(\texttt{nyse} :: \texttt{prices}) = \{\texttt{Ticker}, \texttt{Date}, \texttt{Measure}, \texttt{Price}\}$
$tup(\texttt{nyse} :: \texttt{prices}) = \{\langle ibm, 10\text{—}27\text{—}97, open, 63.67\rangle, \ldots\}$

$conc(\texttt{nyse} :: \texttt{prices}) = \texttt{prices}$
$conc(\texttt{Ticker}) = \texttt{Ticker}$
$rel(\texttt{bse}) = \{\texttt{bse} :: \texttt{prices}\}$
$col(\texttt{bse} :: \texttt{prices}) = \{\texttt{Date}, open\_ibm, open\_ms, ...\}$
$conc(open\_ibm) = \texttt{Price}$
$crit(open\_ibm) = \{\texttt{Measure}, \texttt{Ticker}\}$
$val(open\_ibm, \texttt{Measure}) = open$
$val(open\_ibm, \texttt{Ticker}) = ibm$
etc.

In an instance, simple columns of relations are denoted as in the classical relational model, while complex columns are of the form ($\texttt{concept FOR criteria} = \vec{v}$), where criteria is a list of criteria and $\vec{v}$ is a tuple of values of the appropriate type for the criteria. In formal definitions, such complex columns are denoted as $(\texttt{concept}, t_{\texttt{criteria}})$, where $t_{\texttt{criteria}}$ is the tuple that maps criteria to $\vec{v}$. Sometimes, $t_{\texttt{criteria}}$ are referred to as *criteria-tuple*. A similar remark applies for complex relations.

The concepts and criteria associated with labels are typically *not* recorded in real-life federations. However, *intuitively*, it can be understood that the concept associated with the label "low" is Price and that the only associated criterion is Measure. In the sequel, the formal notion of instances defined above shall be referred to as *abstract instances* to distinguish them from the "real" (i.e. real-life) instances, defined shortly. For an abstract instance to be a legal instance of a federation scheme, certain consistency conditions should be met.

Definition 2.1.3 (Legal Instances) *Let $\mathcal{I}$ be an abstract instance of a federation scheme $\mathcal{S}$. Then $\mathcal{I}$ is said to be a legal instance provided it satisfies the following conditions.*

1. *The following sets are pairwise disjoint: each set of relation id's associated with a given database, each set of column id's associated with a given relation.*

2. *Whenever $a, b \in col(r)$, $a \neq b$, and both a, b correspond to complex attribute names, i.e. $crit(a) \neq \emptyset \neq crit(b)$, it is required that $crit(a) = crit(b)$. In words, the criteria sets associated with any two complex columns in a relation must be identical.*

3. *For each relation id r, for each tuple $t \in tup(r)$, for $a \in col(r)$, it is required that $t[a] \in dom(conc(a))$, i.e. the relations must respect the types of the concepts*

associated with their column labels.

4. *For $a \in col(r) \cup rel(d)$, r being any relation id, and d being any database in $\mathcal{D}$, and $N \in crit(a)$, it is required that $val(a, N) \in dom(N)$, i.e. the values associated with criteria should belong to the appropriate domains.*

In the sequel, references to abstract instances should be understood as references to legal (abstract) instances. The first condition simply ensures that the id's associated with columns and relations are unique. Condition 2 ensures that a fixed set of attribute domains are placed along the row dimension, thus making the 3-dimensional representation of information consistent. Conditions 3 and 4 simply say that the instance respects attribute types.

2.2 Real Federations and Federation Model Bridged

The notion of abstract instances defined in Definitions 2.1.2 and 2.1.3 makes the idea of (legal) instances in the federation model precise. In addition, it also makes the notion of a 3-dimensional representation of data containing several logical dimensions (attributes) precise. However, the following questions arise: (1) How can real-life federations be captured in the formal framework? (2) How relevant is the formal notion of abstract federation instances to practice, and specifically, for the purpose of interoperability? Let us deal with question 1 first, by defining real instances.

Definition 2.2.1 (Real Instance) *A real instance $\mathcal{F}$ of a federation scheme $\mathcal{S}$ is simply a named collection of relational databases such that: (i) $\mathcal{F}$ contains a database corresponding to each database name d in $\mathcal{S}$; (ii) each simple (resp., complex) relation name R associated with a database d in $\mathcal{S}$ corresponds to a relation label (resp., set of relation labels) in $\mathcal{F}$; (iii) each simple (resp., complex) attribute name A associated with a relation name R in database d in $\mathcal{S}$ corresponds to a column label (resp., set of column labels) in $\mathcal{F}$; (iv) all relation labels corresponding to a relation name R have the same set of associated column labels.*

Given an abstract instance $\mathcal{I}$ of a federation scheme $\mathcal{S}$, it is straightforward to construct a real instance $\mathcal{F}$ by turning the various id's in $\mathcal{I}$ into labels. Such a real instance $\mathcal{F}$ is called the real instance *corresponding* to the abstract instance $\mathcal{I}$. The federation shown in Figure 1 is indeed the real instance of the federation scheme $\mathcal{S}_1$, corresponding to the abstract instance sketched following Definition 2.1.2. Notice that (i) the notions of concepts and criteria are not present in the definition of a real instance; (ii) there is no constraint on the labels chosen for the relations or columns. Indeed, in real-life federations, users most often have total control over the chosen labels, and the concept and criteria information may not be explicitly present. Thus, the notion of real instances captures real-life federations.

Let us next address question 2 above. Abstract and real instances can be connected by treating the various labels in the real instance as though they were id's. The actual concepts and criteria associated with them, *which are not explicitly present*, can be attached in a *non-intrusive way* in the form of system catalog tables, formalised next.

Definition 2.2.2 (Catalog Database) *The catalog database associated with an abstract instance $\mathcal{I}$ consists of the following three relations (which are called catalog tables):* `dbscheme(db, relid, rel_label, rel_concept)`, `relschemes(relid, attrid, attr_label, attr_concept)`, `criteria(id, criteria, value)` *satisfying the following conditions.*

- *the relation* `dbscheme` *contains a tuple (d, r, ℓ, c) exactly when, according to $\mathcal{I}$, database d has a relation with relation id r whose label is ℓ and underlying concept is c.*
- *the relation* `relschemes` *has a tuple (r, a, ℓ, c) exactly when, according to $\mathcal{I}$, relation with id r has attrid a as one of its associated attributes, ℓ is the label of a while c is its underlying concept.*
- *the relation* `criteria` *has a tuple (i, cr, v) exactly when, according to $\mathcal{I}$, the id i has cr as one of its criteria which has the associated value v.*

The catalog database associated with the federation of Figure 1 is shown in Figure 2.

The database `catalog` can be treated as a distinguished database from a formal viewpoint in that it always consists of the three catalog tables defined above. *Let us stress that casual users do not have to explicitly manipulate the catalog db.* For linking an abstract instance to its corresponding real instance, the notion of an augmented instance is proposed and defined next. Let $\mathcal{F}$ be a real instance corresponding to an abstract instance $\mathcal{I}$. The *augmented instance* associated with $\mathcal{F}$ and $\mathcal{I}$ means the federation obtained by adding to $\mathcal{F}$ the distinguished database `catalog`, the catalog database associated with $\mathcal{I}$. Our first result is that there is a one-to-one correspondence between the (legal) abstract instances of a federation scheme and augmented

db	relid	rel_label	rel_concept
nyse	r_1	prices	prices
tse	r_2	quotes	prices
bse	r_3	prices	prices
mse	r_4	ibm	prices
mse	r_5	ms	prices
mse	...	...	...

dbschemes

relid	attrid	attr_label	attr_concept
r_1	a_1	Ticker	Ticker
...	...	...	...
r_3	a_i	open_ibm	Price
...	...	...	...
r_4	a_j	low	Price
...	...	...	...

relschemes

id	criteria	value
r_4	Ticker	ibm
r_5	Ticker	ms
a_i	Measure	open
a_i	Ticker	ibm
...	...	...
a_j	Measure	low
...	...	...

criteria

Figure 2: The catalog database associated with the federation of Figure 1.

real instances.

Theorem 2.2.1 *Let $\mathcal{S}$ be a federation scheme. Then to every abstract instance of $\mathcal{S}$, there exists an equivalent (augmented) real instance of $\mathcal{S}$, and vice versa.*

PROOF:
For a given abstract instance $\mathcal{I}$, a simple encoding scheme lets us create an equivalent augmented real instance.

First, we construct a real instance corresponding to $\mathcal{I}$ in the following manner:

- For each id in $\mathcal{I}$, we choose a new label that has never been used before;
- The real instance has a database $d \iff d \in \mathcal{D}$;
- $\forall d \in \mathcal{D}$, database d in the real instance has one relation corresponding to each relation id $r \in rel(d)$;
- Let r be a relation id $\in rel(d)$ and let $real(r)$ be the corresponding relation in the real instance. Then, $real(r)$ has one column corresponding to each column id $c \in col(r)$;
- Let r be a relation id $\in rel(d)$ and let $real(r)$ be the corresponding relation in the real instance. Let c be a column $\in col(r)$ and let $real(c)$ be the corresponding

column in the real instance. Then, $\forall$ tuple $t \in tup(r)$ $real(r)$ has a tuple $real(t)$ s.t. $real(t)[real(c)] = t[c]$

Then, we can build the catalog tables for that real instance in the following manner:

- for each database d_i in the abstract instance, for each relation id $r_{i,j} \in rel(d_i)$, a tuple $\langle d_i, r_{i,j}, r_{i,j}, conc(r_{i,j})\rangle$ is added to table `dbschemes`;
- for each relation id r_i in the abstract instance, for each column id $c_{i,j} \in col(r_i)$, a tuple $\langle r_i, c_{i,j}, c_{i,j}, conc(c_{i,j})\rangle$ is added to table `relschemes`;
- for each relation id r_i in the abstract instance, for each criterion $k_{i,j} \in crit(r_i)$, a tuple $\langle r_i, k_{i,j}, val(r_i, k_{i,j})\rangle$ is added to table `criteria`;
- for each column id c_i in the abstract instance, for each criterion $k_{i,j} \in crit(c_i)$, a tuple $\langle c_i, k_{i,j}, val(c_i, k_{i,j})\rangle$ is added to table `criteria`;

It is obvious that the above construction preserves information.

Going from real augmented instance to abstract instance is simply done by using the converse of this scheme, which means defining the set $\mathcal{D}$ and the various functions *rel*, *col*, *tup*, *crit*, *conc* and *val* such that they agree with the entries in the catalog tables and the content of the real instance. E.g., in the abstract instance, $tup(r)$ will have a tuple $\Longleftrightarrow$ the corresponding relation in the real instance has that tuple.

Again, the fact that the information is preserved should be obvious. In particular, the mappings in either direction do not map two instances whose information is not equivalent to the same target instance. From this the one-to-one correspondence follows. ∎

Incorporating data marts: So far, attention has been focused on relational databases. Many data marts (like data warehouses) that are based on the so-called ROLAP approach adopt a star schema or a snowflake schema for their implementation. Let us call such data marts relational data marts. It is easy to see that such schemas correspond to federated schemas where both relation names and attributes are simple. Thus, the notions of a federation scheme and instance defined in Definitions 2.1.1 and 2.1.2 subsume relational data marts.

Chapter 3

The Syntax of nD-SQL

In this chapter, we present the nD-SQL language, a query language downward compatible with SQL and which takes advantage of the Federation Model to query, restructure and aggregate data. This language also lets users express queries asking for the computation of arbitrary sets of group-bys and/or multiple visualisations of the same results. This thesis covers only the non-nested, querying fragment of nD-SQL.

Chapter 3 presents the syntax of nD-SQL by explaining the additions made to SQL. The semantics of nD-SQL will be illustrated with examples. For a formal description of the semantics, refer to Chapter 4. The complete syntax of nD-SQL in the form of a grammar is given in the Appendix.

The first part of this chapter explains the syntactic extensions for multi-dimensionality and restructuring. The second part will present the extensions regarding multiple granularity aggregations and multiple renderings of the same query result. *Throughout the thesis, the federation of Figure 1 will be used as a running example to illustrate the nD-SQL queries.*

3.1 Multi-dimensionality and Restructuring

3.1.1 nD-SQL Syntax

nD-SQL uses the classic SELECT, FROM, WHERE, GROUP BY and HAVING clauses of SQL, but adds to the syntax in several manners. Table 1 summaries the syntactic additions and we refer the reader to that table for details on the following points.

(1) FROM clause: In addition to declaring the usual tuple variables (called 'aliases' in SQL), users can now also declare variables ranging over *database names, a set of*

relations, or *a set of columns of relation(s)*. These new variable types are inspired by those used with the SchemaSQL language ([LSS96]). The syntax of variable declarations is summarised in Table 1. In that table, `db` and `rel` can be either constants or variables of the appropriate kinds.

(2) WHERE clause: *n*D-SQL introduces two new interpreted constraints (in Table 1) which may be used in the WHERE clause to constrain relation or column variables to range over a "homogeneous" set of schema objects, i.e. over relations/columns having the same concept and set of criteria. The use of such constraints will help ensure queries are "well-typed", a notion that will be formally defined in Section 3.1.3. In Table 1 (constraining variables), *var* can be a rel_var or a col_var.

Syntax for...	New Element of Syntax		
declaring variables in FROM clause	Variable Type	Declaration Syntax	Variable ranges over...
	db_var	-> var	the names of the dbs in the federation
	rel_var	db -> var	the relations in database(s) db
	col_var	db::rel -> var	the columns of the relation(s) rel in database(s) db
	tuple_var	db::rel var	the tuples of the relation(s) rel in database(s) db
constraining variables in WHERE clause	Constraint	Syntax	Variable constrained to range over...
	ISA condition	*var* ISA *concept*	objects representing concept *concept*
	HASA condition	*var* HASA *criterion*	objects having criterion *criterion* in their criteria set
extracting domain values in SELECT, WHERE, GROUP BY and HAVING clauses	Domain Type	Values in domain	
	db_var	database names db_var ranges over	
	tuple_var.attribute	values of attribute *attribute* in the tuples tuple_var ranges over	
	col_var.criterion	values of criterion *criterion* of the columns col_var ranges over	
	rel_var.criterion	values of criterion *criterion* of the relations rel_var ranges over	
	tuple_var.col_var	values of concept concept(col_var) under each column col_var ranges over in the tuples tuple_var ranges over	
creating complex columns in SELECT clause	*domain*$_0$ [AS *label*] FOR (*domain*$_1$ {, *domain*$_i$}), $i > 1$		
creating complex relations in SELECT clause	SELECT (*select_objects_list*) [AS *label*] FOR *domain*$_1$ {, *domain*$_i$}, $i > 1$		

Table 1: New elements of syntax in *n*D-SQL

As an example of the use of variable declarations and of proper constraints, here is what the FROM and WHERE clauses could contain in order to query the data from Figure 1(d):

```
FROM     mse -> R, mse::R T, mse::R -> C
WHERE    R HASA Ticker AND C ISA Price
```

Note how the rel_var R *is restricted to range over the relations of database* mse *having* Ticker *values as criteria values, and how the col_var* C *is restricted to range over the columns of these relations having* Price *values as their underlying concept.*

(3) Since SQL allows only tuple variables, it has only one type of domain expression, expressions of the form tuple_var.attr (abbreviated as attr). In addition to this, nD-SQL also allows the domain expressions db_var, tuple_var.col_var and V.criterion, where V is a relation (or column) variable and criterion is one of the criteria of the relations (or columns) the variable ranges over. The expression db_var extracts the names of the databases in the federation, the next expression extracts values of the concept in complex columns, while the last expression is used to extract criteria values from the federated relation schema. All of these domain expressions can be used in the SELECT and GROUP BY clauses, and in conditions in the WHERE and HAVING clauses.

Also, the *underlying concept* of a domain is defined as follows:

Definition 3.1.1 (Underlying concept of a domain)

$$\text{undconc}(\mathit{domain}) = \begin{cases} \texttt{db\_var} & \text{if domain is of the form } \texttt{db\_var} \\ \texttt{attribute} & \text{if domain is of the form } \texttt{tuple\_var.attribute} \\ \texttt{criterion} & \text{if domain is of the form } \texttt{rel\_var.criterion} \\ \texttt{criterion} & \text{if domain is of the form } \texttt{col\_var.criterion} \\ \mathit{conc}(\texttt{col\_var}) & \text{if domain is of the form } \texttt{tuple\_var.col\_var} \end{cases}$$

where the concept of a complex column over which a col_var ranges is referred to as *conc(col_var)*. In the rest of this thesis, the set of criteria associated with a column or relation variable var is referred to as *crit(var)*.

As an example of the use of each kind of domains, the following query "flattens" the data from the tables of Figure 1(d) into a form similar to table nyse::prices:

```
        SELECT   R.Ticker, T.Date, C.Measure, T.C AS Price
(Q1)    FROM     mse -> R, mse::R T, mse::R -> C
        WHERE    R HASA Ticker AND C ISA Price
```

Note in this query, in addition to the use of the HASA/ISA *conditions to constrain the relation and column variables, the extraction of the values of criteria* C.Measure *into a column of its own. The multiple columns that* C *ranges over are aligned into a single column by the select_object* T.C AS Price. *Here, each tuple of each table of Figure 1(d) is broken down into many output tuples, one per value of the criterion* Measure.

(4)In order to create complex columns and relations, a mechanism is needed in order to deposit data values as criteria values. To deposit data values as column criteria values the following new type of select_objects is to be used in the SELECT clause:

$domain_0$ [AS *label*] FOR ($domain_1$ {, $domain_i$}), $i > 1$

The optional labels use the following syntax: any series of constant strings (in double quotes) or domain expressions among those for the criteria $domain_j, j \geq 1$, concatenated together using the ampersand (&) symbol. Examples of labels when the criteria list ($domain_1$ {, $domain_i$}), $i > 1$ is (T.ticker) could be: "Price for Year = " & T.Ticker, "Price for " & T.Ticker, T.Ticker & "'s Price" or even simply T.Ticker.

If no label (AS sub-clause) is present, then a default should be used. When there is no FOR sub-clause, it is proposed to use the name of the underlying concept (Definition 3.1.1) of $domain_0$ (similar to the SQL convention in the absence of an AS sub-clause). When one FOR sub-clause is present, the proposed default is a comma separated list of the criteria values (equivalent to the label $domain_1$ & "," & $domain_2$ & "," & ...). When more than one FOR sub-clause are present in the SELECT clause, that list of criteria values could be preceded by undconc($domain_0$), the name of the underlying concept of the column.

Relating the syntax and the model: The use of the FOR sub-clause with a select_object indicates that there should be a complex attribute with name: (undconc($domain_0$), {undconc($domain_1$), undconc($domain_2$), ...}) in the output relation schema).

The following example illustrates the use of this syntax by transforming the content of nyse::prices into a format similar to the one of table tse::quotes.

```
      SELECT   T.Ticker AS Stocks, T.Date, T.Price AS T.Measure FOR T.Measure
(Q2)  FROM     nyse::prices T
```

Note in this query how the multiple Price columns are created, one for each

`Measure` *values. by the use of the* `FOR` *sub-clause. Note also how these* `Measure` *values are used as column labels. This representation of data is an example of what is commonly called the* cross-tab *representation.*

(5)To deposit data values as relation criteria, the select_objects of the SELECT clause are enclosed in parentheses and an outer `FOR` sub-clause is applied:

`SELECT (select_objects_list) AS` *label* `FOR` $domain_1$ `{,` $domain_i$`}`, $i > 1$

Relating the syntax and the model: The use of the outer `FOR` sub-clause indicates that a relation with name (rel_k, {undconc($domain_1$) , undconc($domain_2$) , ...}) should be created. The relation concept rel_k for the output relations should be system-generated in order to prevent conflicts with other relation concepts in the catalog.

The following example illustrates the creation of complex relations, while an aggregation is performed.

```
      SELECT   (Avg(T.C) AS "AvgPrice FOR Measure = " & C.Measure FOR C.Measure)
                    AS T.Date FOR T.Date
(Q3)  FROM     bse::prices -> C, bse::prices T
      WHERE    C ISA Price
      GROUP BY C.Measure, T.Date
```

This query takes the aggregation of each individual `Price` *for a given* `Measure` *on a given* `Date` *(i.e. the aggregation is over* `Tickers`*). Here, note that the aggregation is performed over a subset of the criteria of* `C`*. The aggregation is performed on* `T.C` *(i.e.* `Price` *values), grouping by* `C.Measure` *(extracting the values of* `Measure`*) and* `T.Date`*. The inner* `FOR` *sub-clause restructures the averages into multiple columns, one per value of* `Measure`*, while the outer* `FOR` *sub-clause restructures the result into multiple relations, one per value of* `Date`*. The result of the query is shown in Figure 3, where the output relations are assumed to be temporarily viewed as members of a database named "output".*

AvgPrice FOR Measure = open	AvgPrice FOR Measure = close	...
50.68	52.87	...

output::10|27|97

...

AvgPrice FOR Measure = open	AvgPrice FOR Measure = close	...
5905K	6308K	...

output::11|01|97

...

Figure 3: Result of query Q3

3.1.2 Allowable Abbreviations

Various abbreviations are acceptable in our syntax. All the abbreviations mentioned here can be used in the SELECT clause.

(1)The abbreviation V.*, for a rel/col_var V, is a shorthand for the enumeration V.*criterion*$_1$, ..., V.*criterion*$_n$, where crit(V) = {*criterion*$_1$, ..., *criterion*$_n$}. This can *also* be used in the GROUP BY clause in aggregating queries;

(2)The abbreviation T.*concept*, for a tuple_var T, used as a select_object in the SELECT clause, is equivalent to the expression T.C FOR C.*, C being a col_var declared over the same relation(s) and with underlying concept *concept*. This abbreviation says to select each instance of a complex column as is, without restructuring;

(3) The abbreviation T.*, for a tuple_var T, used as a select_object, says to select *all* columns of the relation that T ranges over, as in classical SQL. As an example of all these abbreviations, query Q4 selects *all* the columns of relation bse::prices. Here, Q4a uses the simplest abbreviation, Q4b and Q4c are intermediary equivalent queries, and Q4d is the fully expanded, explicit query equivalent to the other three.

```
      SELECT T.*                   SELECT T.Date, T.C FOR C.*
(Q4a) FROM   bse::prices T   (Q4c) FROM   bse::prices T, bse::prices -> C
                                   WHERE  C ISA Price

      SELECT T.Date, T.Price       SELECT T.Date, T.C FOR (C.Measure, C.Ticker)
(Q4b) FROM   bse::prices T   (Q4d) FROM   bse::prices T, bse::prices -> C
                                   WHERE  C ISA Price
```

(4)Suppose the same aggregation is to be performed *individually* on each column of a relation that "is a" concept. Then the abbreviation AGG(T.concept) can be used instead of using the select_object AGG(T.C) FOR C.* and having to explicitly declare and constrain the column variable. Example Q5a exemplifies the use of this abbreviation by querying bse::prices and taking the average of the Prices throughout the Dates for *each* Measure and Ticker. Q5b is the equivalent explicit query.

```
      SELECT  AVG(T.Price)         SELECT  AVG(T.C) FOR (C.Measure, C.Ticker)
(Q5a) FROM    bse::prices T  (Q5b) FROM    bse::prices T, bse::prices -> C
                                   WHERE   C ISA Price
```

Note that we will use the term *explicit query* to denote a query for which all abbreviations are expanded.

3.1.3 Well Typing

Intuitively, a query can be meaningful only if it maps legal instances to legal instances. More precisely, the following definition is used.

Definition 3.1.2 (Well-Typing) *An* n`D-SQL` *query* `Q` *is well-typed provided for every legal instance* $\mathcal{I}$, `Q`$(\mathcal{I})$, *viewed as an instance is also legal.*

Ensuring well-typing is important for query processing, not only to make sure the result presented to the user is meaningful, but also for ensuring aggregations can be correctly applied. Thus, an efficient algorithm for testing well-typing is essential. In order to develop such an algorithm, we define a syntactic notion stronger than well-typedness, *well-formedness*:

Definition 3.1.3 (Well-Formedness) *An* n`D-SQL` *query* `Q` *is well-formed provided that it fulfills the following conditions:*

(i) each relation variable is restricted (by `ISA` *and* `HASA` *conditions) to range over relations having the same concept and criteria set;*

(ii) each column variable is restricted (by `ISA` *and* `HASA` *conditions) to range over columns having same concept and same set of criteria;*

(iii) all the complex columns created in the `SELECT` *clause have the same set of criteria;*

As stated above, well-formedness is a stronger notion than well-typedness. Formally:

Theorem 3.1.1 *If a query is well-formed it is also well-typed.*

<u>PROOF</u>:

Let `Q` be a query. let $\mathcal{I}$ be a legal instance and `Q`$(\mathcal{I})$ the output of `Q` on $\mathcal{I}$.

1. Clearly, condition 1 of legality will be met by construction of `Q`$(\mathcal{I})$, since all id's produced will be unique;

2. Suppose that a and b are two column id's in `Q`$(\mathcal{I})$ that correspond to complex columns. Since `Q`$(\mathcal{I})$ has complex columns it implies that `Q` must involve select objects of the form *domain*$_0$ `[AS` *label*`]` `FOR` *domain_list*. By condition (iii) of

well-formedness, every list of domains *domain_list* in inner FOR sub-clauses will contain the same domains, and thus a and b will have the same criteria set. The query result will thus meet condition 2 of legality.

3. Conditions (i) and (ii) of well-formedness imply that:

   - For each domain expression of the form tuple_var.attr (attr being a constant), the value will come from the domain of attr. The condition on relation variables ensures the tuple_var ranges over relations having identical schemas, and the type of a given concept (attr) in all these relations must be fixed by virtue of the legality of $\mathcal{I}$;
   - For each domain expression of the form tuple_var.col_var, since col_var is restricted to range over columns with the same concept, the same argument as above can be made;
   - By virtue of the legality of $\mathcal{I}$, the condition on relation variables ensures that, for each domain expression of the form rel_var.crit, the value comes from the domain of criterion crit;
   - By virtue of the legality of $\mathcal{I}$, the condition on column variables ensures that, for each domain expression of the form col_var.crit, the value comes from the domain of criterion crit.

   Since the arguments above apply to each select expression appearing in the SELECT clause, we conclude that condition 3 of legality has to be met by Q($\mathcal{I}$).

4. Since the arguments from 3 above also apply to each expression appearing in a criteria position in a FOR sub-clause, we conclude that condition 4 of legality is also met by Q($\mathcal{I}$).

∎

Theorem 3.1.1 immediately yields a sufficiency test for testing well-typing: test whether the query satisfies the conditions for being well-formed. We can test the latter in time linear in the size of a given query, provided the catalog tables are properly indexed. The algorithm veryfying a query is well-formed is presented in Figure 4. The semantics of nD-SQL (presented in the next chapter) is defined for well-formed queries, and thus not for all well-typed queries (see section 8.2 for an example of a query which is well-typed but not well-formed and that is thus not supported by the current semantics). A more complex semantics would be necessary to support all well-typed queries, and developping such a semantics is part of our ongoing work.

```
INPUT: An nD-SQL query
       The catalog database for the federation being queried

OUTPUT: A boolean value, true if query is well-formed, false if not


still_good = true;
first_for_sub-clause = true;
for each select_object in the SELECT clause do
    if the select_object has a FOR sub-clause then
        if first_for_sub-clause then
            store crit_list in variable criteria;
            first_for_sub-clause = false;
        else
            if crit_list is not equivalent to criteria then
                still_good = false;
            end if
        end if
    end if
end for

if still_good then
    for each condition in the WHERE clause do
        if the condition is an ISA or an HASA condition then
            associate the condition with the proper variable;
        end if
    end for

    for each variable declaration in the FROM clause do
        if still_good AND the declared variable is a rel_var or a col_var then
            if the range of the variable contains a non-instantiated variable then
                delay the check for this variable;
            else
                using the range of the variable + the associated ISA and HASA conditions,
                query the catalog database to instantiate the variable, also getting each instance's
                concept and criteria set;
                if all instances do not have same concept and criteria set then
                    still_good = false;
                end if
            end if
        end if
    end for
end if

if still_good then
    return true;
else
    return false;
end if
```

Figure 4: Algorithm for verifying if a query is well-formed

The algorithm first checks if the criteria set in every FOR sub-clause is the same. If so, it then associates each ISA and HASA condition in the WHERE clause to the proper variable and verifies for each rel_var and col_var declared in the FROM clause that those conditions associated with it, combined with the variable's declared range, are sufficient to restrict it to range over objects having same concept and same criteria set. This ensures that all three conditions for well-formdness are met.

As an example of the use of the algorithm, query Q1 is found to be well-formed in the following manner: (1) since there is no inner FOR sub-clause in the SELECT clause the query passes the first test; (2) the condition R HASA Ticker is associated with the rel_var R and the condition C ISA Price is associated with the col_var C; (3) using the range of R and the condition R HASA Ticker, the catalog database is queried and all returned instances for R have same concept (prices) and criteria set ({Ticker}). All possible instances for C are also found to have same concept and criteria set. The three conditions for well-formdness are thus met.

3.2 Enhancing nD-SQL for OLAP: Multiple Visualisations and Arbitrary Sets of Group-bys

Since the proposal by Gray et al. [Gray+96] for the powerful CUBE operator, researchers have developed several efficient algorithms for computing this expensive operator [Agar$^+$96, ZDN97]. The CUBE operator corresponds to aggregation at exponentially many granularities. It has been recognised [Agar$^+$96, ZDN97] that in practice, a user may be interested in specific subsets of group-bys. Two such examples are ROLLUP (e.g., {{Date, Ticker}, {Date}, {}}) and its converse DRILLDOWN. While these operators are important, in general, and depending on the application at hand, users may be interested in subsets that need not be covered by these operators. See Example 3.2.4 for one such "interesting" subset. In this section, some simple extensions to nD-SQL are developed which lead to a powerful mechanism for expressing arbitrary subsets of group-bys. In addition, together with the restructuring capabilities of nD-SQL, these extensions allow for the computation of arbitrary multiple granularity aggregations and the visualisation of the results in multiple ways.

Following OLAP terminology, each of the names in a federation scheme is referred to in the sequel as a *logical dimension.* Each variable declared in the FROM clause of an nD-SQL query can thus be associated with a set of logical dimensions, as follows:

Definition 3.2.1 (Logical Dimensions associated with a variable) *Let Q be an* nD-SQL *query and let* V *be a variable declared in the* FROM *clause of Q. The set of* *logical dimensions associated with* V *is:*

$\{\mathtt{D}\}$,	*if* D = V *is a database variable,*
$\{\mathtt{V}.crit_1, \mathtt{V}.crit_2, ..., \mathtt{V}.crit_p\}$,	*if* V *is a column or relation variable, having criteria* $crit_i$, $i \leq p$
$\{\mathtt{T}.simp_1, \mathtt{T}.simp_2, ..., \mathtt{T}.simp_u, \mathtt{T}.\mathtt{C}_1, ..., \mathtt{T}.\mathtt{C}_s\}$,	*if* T = V *is a tuple variable, the relations over which* T *ranges have* u *simple columns with concepts* $simp_i, i \leq u$ *and* s *complex columns for which a column variable* $\mathtt{C}_j$ *is declared,* $j \leq s$

Definition 3.2.2 (Logical Dimensions associated with a query) *Let* Q *be an* nD-SQL *query. The logical dimensions associated with* Q *is the set of logical dimensions associated with the variables declared in the* FROM *clause of* Q.

3.2.1 Extensions to the nD-SQL Syntax

The enhancements to the syntax of nD-SQL permitting the expression of multiple sub-aggregates and visualisations are summarised in Table 2. The main addition is a new kind of variable called *dimension variable*, ranging over the *names* of all logical dimensions associated with the query, except those being aggregated. An nD-SQL query Q with dimension variables still maps a federation to a set of relations. However, for such a query we define the mapping in the following manner: The result of Q is the same set of relations as the combined result of *the set of* nD-SQL *queries* without *dimension variables*, obtained by instantiating the dimension variables in Q to all possible dimension names that satisfy the constraints on the dimension variables, specified in the WHERE clause of Q. Here is an extremely simple example to illustrate the ideas.

Example 3.2.1

```
            SELECT    X, SUM(T.Price)
(Q7)        FROM      nyse::prices T, DIM X
            GROUP BY  X
```

The only dimension variable is X. *The only non-dimension variable declared in the* FROM *clause is* T, *whose associated dimensions are* T.Date, T.Ticker, T.Measure *and* T.Price. *Of these,* T.Price *is being aggregated. So, the dimension variable* X *ranges over the dimension names* T.Date, T.Ticker, *and* T.Measure. *The equivalent set of queries without dimension variables are as follows.*

```
        SELECT    T.Ticker, SUM(T.Price)            SELECT    T.Date, SUM(T.Price)
(Q7a)   FROM      nyse::prices T            (Q7b)   FROM      nyse::prices T
        GROUP BY  T.Ticker                          GROUP BY  T.Date
```

```
        SELECT    T.Measure, SUM(T.Price)
(Q7c)   FROM      nyse::prices T
        GROUP BY  T.Measure
```

Thus, this query expresses the aggregation of T.Price with respect to each of the three possible group-bys – T.Ticker, T.Date, *and* T.Measure, *which corresponds to the level-1 slice of a* CUBE. ■

Syntax for...	New Element of Syntax		
declaring variables in FROM clause	Variable Type	Declaration Syntax	Variable ranges over...
	dim_var	DIM var	logical dimensions associated with query, except those being aggregated
constraining variables in WHERE clause	Constraint	Syntax	Effect
	rel-ops	var Op dimension, (Op being one of $=$, $\leq$, ...)	var must satisfy the relationship Op w.r.t. dimension
		var_1 Op var_2	instantiations of var_1 and var_2 must satisfy the relationship Op
	membership	var IN set of dimensions	instantiation of var must belong to the set of dimensions specified
		var NOT IN set of dimensions	instantiation of var must not belong to the set of dimensions specified
	special	var CAN BE NONE	range of var includes the special constant NONE
abbreviations in SELECT clause	Description	Syntax	Equivalent to
	Shorthand for declaring multiple dimension variables	DIM X_1, X_2, ..., X_n	DIM X_1, DIM X_2, ..., DIM X_n
abbreviations in WHERE clause	Description	Syntax	Equivalent to
	Shorthands for constraints on all dimension variables	DIMS CAN BE NONE	X_1 CAN BE NONE AND ... X_n CAN BE NONE
		DIMS IN set of dimensions	X_1 IN set of dimensions AND ... AND X_n IN set of dimensions
	Shorthand for transitive constraints	$X_1 < X_2 < X_3$	$X_1 < X_2$ AND $X_2 < X_3$

Table 2: Syntax for dimension variables; $X_1, \ldots, X_n$ are all the dim_vars declared in the query.

Conceptually, the nD-SQL query Q7 can be thought of as producing as output the three relations corresponding to the results of queries Q7a–c. An alternative way to think about it is that it produces the union of the three relations mentioned above. In forming such a union, one can adopt Gray et al.'s approach of using the special value "All" to correctly represent the union.

Constraints on dimension variables include the standard rel-ops $=, \leq, <, >, \geq, \neq$. They are interpreted w.r.t. the lexicographic ordering of the dimension names. E.g., T.Date < T.Ticker. Such constraints may be used to eliminate duplicate group-bys from being presented in the result, similar to the use of the DISTINCT keyword in the SELECT clause. Also useful will be a special constant, NONE, inspired by the special constant all introduced by Gray et al. [Gray+96].[1] The constant NONE is given a special status w.r.t. the way the rel-ops are interpreted. *It is assumed that* NONE *Op* NONE *is always true for all rel-ops Op; furthermore, it is assumed that* $\langle$*dimension*$\rangle$ < NONE *is always true, for all dimension names* $\langle$*dimension*$\rangle$. Besides rel-ops, constraints involving the IN operator are also allowed, with the obvious semantics.

Finally, a special type of constraint is introduced using which a dimension variable is allowed to assume the value NONE. This feature is particularly useful for specifying multiple granularity aggregations, as several examples will show. Table 2 also explains the abbreviations allowed in nD-SQL.

Example 3.2.2 *Let us now revisit the previous example and see how a* CUBE *of* Price *values over the dimensions* T.Ticker, T.Date *and* T.Measure *can be expressed.*

```
             SELECT   X, Y, Z, SUM(T.Price)
             FROM     nyse::prices T, DIM X,Y,Z
(Q8)         WHERE    X < Y < Z AND DIMS CAN BE NONE
             GROUP BY X, Y, Z
```

In this query, X, Y *and* Z *can each range over the dimension names* {T.Ticker, T.Date, T.Measure, NONE}. *The condition* X < Y < Z *(an abbreviation for* X < Y AND Y < Z*) further restricts the possible groupings, thus the only groupings done will be as shown in Figure 5. This query computes exactly the* CUBE *operator over the previously mentioned dimensions. Note that the condition* X < Y < Z *prevents the same result from being presented more than once. Without that condition, the query could be instantiated in one case with* X = T.Ticker, Y = T.Date *and* Z = NONE *and in another with* X = T.Date, Y = T.Ticker *and* Z = NONE. *Both groupings are simply structural permutations of the same logical result.*

Finally, note that if the user wants to express ROLLUP, *in place of a* CUBE, *then all s/he needs to do is modify the constraints on dimension variables to:* X IN {T.Date,

[1]The name NONE seems more appropriate for the use made of this constant here.

NONE} AND Y IN {T.Measure, NONE} AND Z IN {T.Ticker, NONE} AND X < Y < Z. *The reader can easily verify that this will produce exactly the* group-bys {T.Date, T.Measure, T.Ticker}, {T.Date, T.Measure}, {T.Date}, *and* {}. ■

X	Y	Z	group by...
T.Date	T.Measure	T.Ticker	T.Date, T.Measure and T.Ticker
T.Date	T.Measure	NONE	T.Date and T.Measure
T.Date	T.Ticker	NONE	T.Date and T.Ticker
T.Measure	T.Ticker	NONE	T.Measure and T.Ticker r
T.Date	NONE	NONE	T.Date
T.Measure	NONE	NONE	T.Measure
T.Ticker	NONE	NONE	T.Ticker
NONE	NONE	NONE	nothing

Figure 5: Groupings done by query Q8

The next example shows the interplay between multiple granularity aggregation and restructuring.

Example 3.2.3

```
                    SELECT   X, AVG(T.Price) FOR Y
(Q9)                FROM     nyse::prices T, DIM X, Y
                    WHERE    DIMS IN {T.Date, T.Measure, T.Ticker}
                             AND X <> Y
                    GROUP BY X, Y
```

This query generates all possible groupings of AVG(T.Price) *along two logical dimensions among* T.Date, T.Measure *and* T.Ticker. *Furthermore, it restructures each particular grouping in multiple ways along the (physical) row and column dimensions, such that multiple visualisations of the same data are provided at once. The visualisations are shown in Figure 6. The instantiated queries thus correspond to the following schemas:* output(Date, (AVG(Price), {Measure})) output(Date, (AVG(Price), {Ticker})) output(Measure, (AVG(Price), {Date})) output(Measure, (AVG(Price), {Ticker})) output(Ticker, (AVG(Price), {Date})) output(Ticker, (AVG(Price), {Measure})) ■

The last example in this section illustrates the power of nD-SQL to generate sets of multiple granularity aggregations which do not seem to be obviously expressible using a combination of operators like CUBE and ROLLUP.

Example 3.2.4 *Consider a relation* db::rel(A,B,C,D,E,F,G), *and suppose a user is looking at the result of* SUM(G) *grouped by* A,B,C. *It is very natural for the user to*

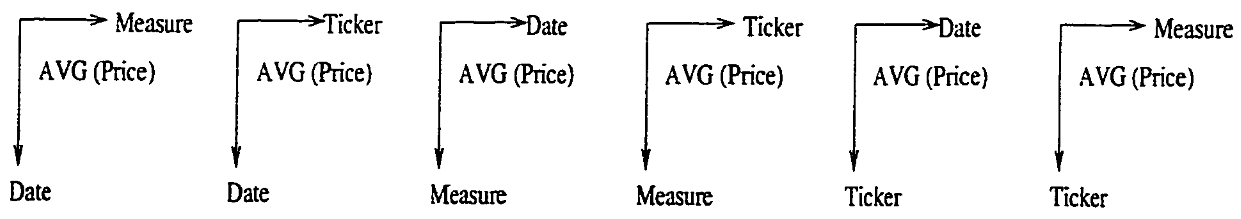

Figure 6: Multiple visualisations resulting from query Q9

want to look at the "neighbourhood" of this group-by*, 1 level below and above* {A,B,C} *in the* group-by *lattice. Specifically, the user might be interested in examining the* group-bys {A,B,C,D}, {A,B,C,E}, {A,B,C,F} *(each of which contains exactly one extra dimension and is one level above* {A,B,C}*) and the* group-bys {A,B}, {A,C}, *and* {B,C} *(which are one level below* {A,B,C} *and are adjacent to it in the cube lattice).*

This query can be expressed as follows.

```
        SELECT W, X, Y, Z, SUM(G)
(Q10)   FROM db::rel T, DIM W,X,Y,Z
        WHERE W < X < Y < Z AND W IN {A,B,C} AND
              X IN {A,B,C} AND Y IN {C, NONE} AND
              Z IN {D,E,F, NONE}
```

Figure 7 depicts the "shape" of this set of group-bys. *It is not clear how such a query can be expressed using known operators.* ■

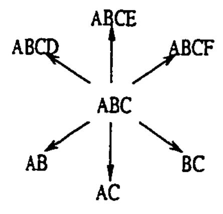

Figure 7: The "neighbourhood" operator

Chapter 4

The Semantics of nD-SQL

[Note: The format of this chapter, and at times the content of section 4.2, follows that of chapter 4 of [S97], with permission of the author]

This chapter presents the semantics of the nD-SQL language. In Section 4.2, the semantics of SQL will be reviewed and in the following section (4.3) the semantics of nD-SQL will be related to that of SQL. But first, a high level, informal description is given.

4.1 Informal Presentation of the Semantics

The SQL language is structured in clauses each having a different purpose. The semantics of SQL is that each table declared in the FROM clause plays the role of a tuple variable that is to be instantiated to some tuple from that table. Each instance tuple for a variable is concatenated with those of the other variables, and the conditions in the WHERE clause are applied to determine whether the concatenated instances satisfy them. In the case of an aggregation query, the satisfying instantiations are then grouped into equivalence classes, some aggregate operations are applied, and one aggregated tuple is produced for each such class. Then, for each of these tuples, the conditions in the HAVING clause are applied. The aggregated tuples satisfying the conditions are then restricted to those columns that are enumerated in the SELECT clause. The restrictions of the (possibly aggregated) tuples so obtained make up the query result.

The major differences between nD-SQL and SQL are that:

1. nD-SQL allows variables of a higher order than tuple variables;

2. An input to a SQL query is a database, whereas an input to an *n*D-SQL query is a federation, i.e. a set of databases;

3. The output of a SQL query is a relation, whereas the output of an *n*D-SQL query is a database, i.e. a set of relations;

4. *n*D-SQL queries data that is structured along 3 physical dimensions but represents *n* logical dimensions;

5. Input databases to *n*D-SQL queries can structure their data along 3 physical dimensions, namely *row, column* and *relation*;

6. Databases output by *n*D-SQL queries may be similarly structured.

These differences are reflected in the semantics in the following manner: In *n*D-SQL, the variables are instantiated to appropriate entries or "objects", corresponding to the 3 physical dimensions. Then, even if the data being queried is structured along those physical dimensions, it actually corresponds to a set of "*logical dimensions*" and we can *model* the data as being "flat" by instantiating each variable to some *logical tuple* over all the *logical dimensions* associated with the variable. Each logical tuple instance for a variable is concatenated with those of the other variables, and the conditions in the WHERE clause are applied to determine whether the concatenated instances satisfy them. In the case of an aggregation query, the satisfying logical tuples are then grouped in equivalence classes, some aggregate operations are applied, and one aggregated tuple is produced for each such class. Then, for each of these tuples, the conditions in the HAVING clause are applied. The aggregated tuples satisfying the conditions are then restricted, and physically structured as dictated by the SELECT clause, which in general may result in a set of output relations.

The following sections will clarify and formalise the above descriptions.

4.2 SQL Semantics Reviewed

We consider the non-nested, querying fragment of SQL.

Throughout this section, query Q12 will be used as a running example to illustrate SQL's semantics. Query Q12 is applied to database nyse.

```
        SELECT   T.Date, T.Ticker, AVG(T.Price)
(Q12)   FROM     prices T
        WHERE    T.Ticker >= 'm' AND T.Ticker < 'n'
        GROUP BY T.Date, T.Ticker
```

A query in SQL assumes a *fixed* scheme for the underlying database, and maps each database to a relation over a fixed scheme, called the *output* scheme associated with the query. Let $\mathcal{D}$ be the set of all database instances over a fixed scheme. Let a query Q be of the form

```
SELECT   attr_list, agg_list
FROM     from_list
WHERE    where_conds
GROUP BY groupby_list
HAVING   have_conds
```

Let $\mathcal{R}$ be the set of all relations over the output scheme of the query Q. The query Q induces a function

$$Q : \mathcal{D} \rightarrow \mathcal{R}$$

from databases to relations over a fixed scheme, defined as follows. Let $D \in \mathcal{D}$ be an input database, and $\mathcal{T}_D$ the set of all tuples appearing in any relation in D. Let τ be the set of tuple variables declared in the FROM clause of Q. We define an *instantiation* as a function $\imath : \tau \rightarrow \mathcal{T}_D$ which instantiates each tuple variable in Q to some tuple over its appropriate range. The conditions where_conds in the WHERE clause induce a boolean function, denoted $sat(\imath, Q)$, on the set of all instantiations, reflecting whether the conditions are satisfied by an instantiation. This is defined in the obvious manner. Let $\mathcal{I}_Q = \{\imath \mid \imath \text{ is an instantiation for which } sat(\imath, Q) = true\}$ denote the set of instantiations satisfying the conditions in the WHERE clause.

The set $\mathcal{I}_{\text{Q12}}$ is shown in figure 8.

T.Date	T.Ticker	T.Price
10|27|97	ms	50.23
10|27|97	ms	48.54
...	...	...
11|01|97	ms	44.60
11|01|97	ms	46.17
...	...	...

Figure 8: The set of instantiations $\mathcal{I}_{\text{Q12}}$ corresponding to query Q12

The query assembles each satisfying instantiation into a tuple for the answer relation, as follows. Let $\mathcal{T}_{\mathtt{attrList}_Q}$ denote the set of all tuples over the scheme `attr_list` such that each value in each tuple appears in the database D. Then the tuple assembly function is a function $tuple_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\mathtt{attrList}_Q}$ defined as follows.

$$tuple_Q(\imath) = \bigotimes_{\text{“}t.A\text{”} \in \mathtt{attr\_list}} \imath(t)[A]$$

Here, the predicate “$t.A$” $\in$ `attr_list` indicates the condition that the attribute denotation $t.A$ literally[1] appears in the list of attributes `attr_list` in the SELECT clause. The symbol $\otimes$ denotes concatenation, and $\imath(t)[A]$ denotes the restriction of the tuple $\imath(t)$ to the attribute A. For an instantiation $\imath$, $tuple_Q(\imath)$ produces a tuple over the attributes `attr_list` listed in the SELECT statement. Suppose Q is a regular query, *i.e.* a query without aggregation. In this case, the `agg_list` is empty, the HAVING and GROUP BY clauses are absent, and the result of the query is captured by the function

$$Q(D) = \{tuple_Q(\imath) \mid \imath \in \mathcal{I}_Q\}$$

To account for aggregation, we need the following extension. We define a relation $\sim$ on the instantiations.

Definition 4.2.1 For $\imath$, $\jmath \in \mathcal{I}_Q$, $\imath \sim \jmath$ iff $\forall$“$t.A$” $\in$ `groupby_list`, $\imath(t)[A] = \jmath(t)[A]$. It is straightforward to see that $\sim$ is an equivalence relation on $\mathcal{I}_Q$. This definition essentially says that two instantiations (satisfying the conditions in the WHERE clause) are $\sim$-equivalent provided they agree on all attributes appearing in the GROUP BY clause.

Let $\mathcal{T}_{\mathtt{agg}_Q}$ denote the set of tuples over the attributes in the `groupby_list`, the aggregate expressions in `agg_list` and the aggregate expressions in the HAVING clause. We define a function $agg_Q : \mathcal{I}_Q \rightarrow \mathcal{T}_{\mathtt{agg}_Q}$ as follows.

$$agg_Q(\imath) = \bigotimes_{\text{“}t.C\text{”} \in \mathtt{groupby\_list}} \imath(t)[C] \quad \bigotimes_{\text{“}agg_B(t.B)\text{”} \in \mathtt{agg\_list} \,\cup\, \mathtt{having\_conds}} agg_B([\jmath(t)[B] \mid \jmath \in \mathcal{I}_Q,\ \jmath \sim \imath])$$

For a given instantiation $\imath$, agg_Q considers all instantiations equivalent to $\imath$, and, for each aggregate operation, say agg_B, indicated on the attribute $t.B$ in the `agg_list` or the HAVING clause, it performs the operation agg_B on the *multiset* of values associated

[1] Modulo the abbreviations permitted in SQL.

with this attribute by any instantiation equivalent to $\imath$. We use $[\ldots]$ instead of $\{\ldots\}$ to denote multisets.

Since only those tuples satisfying the having_conds should be used to obtain the query result, let the set of such tuples be:

$$aggtuple_Q = \{agg_Q(\imath) \mid \imath \in \mathcal{I}_Q \ \wedge \text{ the tuple } agg_Q(\imath) \text{ satisfies the conditions } \texttt{have\_conds} \text{ in the } \texttt{having} \text{ clause}\}.$$

As an illustration, the set $aggtuple_{Q12}$ is shown in figure 9.

T.Date	C.Measure	T.C
10|27|97	open	6367575
10|27|97	close	6368342
10|27|97	low	6360329
...	...	...
11|01|97	open	6367111
11|01|97	close	6368340
11|01|97	low	6366500
...	...	...

Figure 9: The set $aggtuple_{Q12}$ corresponding to query Q12

We may need to discard some extraneous attributes and/or aggregations that are not mentioned in the SELECT clause in order to obtain the query result.

Let $\mathcal{T}_{\texttt{objectList}_Q}$ be the set of tuples over the attributes and aggregates in the attr_list and the agg_list. We define a function $disc_Q : \mathcal{T}_{\texttt{agg}_Q} \rightarrow \mathcal{T}_{\texttt{objectList}_Q}$ as follows:

Given a tuple $\lambda \in aggtuple_Q$, then

$$disc_Q(\lambda) = \bigotimes_{A \in \texttt{attr\_list} \cup \texttt{agg\_list}} \lambda[A]$$

where A is any attribute or aggregate expression appearing in the attr_list or the agg_list.

The query result is then defined as:

$$Q(D) = \{disc_Q(\lambda) \mid \lambda \in aggtuple_Q\}$$

For our example, the query result $Q12(D) = aggtuple_{Q12}$ since there are no attributes or aggregations captured by $aggtuple_{Q12}$ that are not selected in the SELECT clause.

4.3 nD-SQL Semantics

In this section we assume without loss of generality that all abbreviations in the given query have been expanded (i.e. that we are dealing with *explicit* queries).

Throughout this section, query Q13 will be used as a running example to illustrate nD-SQL's semantics.

```
        SELECT    T.Date, SUM(T.C) FOR C.Measure
(Q13)   FROM      bse::prices -> C, bse::prices T
        WHERE     C ISA Price
        GROUP BY T.Date, C.Measure
```

The main difference between nD-SQL and SQL is that nD-SQL allows for querying data structured along the three physical dimensions underlying the relational model (that is row, column and relation).

The semantics defines a query result as a mapping between the federation model's view of the data in terms of logical dimensions, irrespective of the physical structure, and the desired output structure. This mapping covers the satisfaction of WHERE conditions and also covers aggregation.

Let a query Q be of the form

```
SELECT    (simple_col_list, agg_simple_col_list,
           complex_col_list, agg_complex_col_list)
          FOR outer_for_crit_list
FROM      vardec_list
WHERE     isa_and_hasa_conds AND other_where_conds
GROUP BY  groupby_list
HAVING    have_conds
```

where the complex_col_list and the agg_complex_col_list are the select_objects with FOR sub-clauses.

Let $\mathcal{F}$ be the set of all possible federation instances and let $\mathcal{D}$ be the set of all possible federated database instances (as defined in Chapter 2).

Then, the query Q induces a function

$$Q : \mathcal{F} \rightarrow \mathcal{D}$$

informally defined as follows:

Each variable in Q is mapped to an object (database, relation id, column id or tuple) over the appropriate range. Then we introduce the notion of a "logical tuple". A logical tuple consists of a tuple over each of the logical dimensions associated with a query (as per definition 3.2.2). For example, one logical tuple associated with the query Q13 would be $\langle$open, ibm, 10|27|97, 63.67$\rangle$ over the dimensions {`C.Measure`, `C.Ticker`, `T.Date`, `T.C`}. Since to each variable's instance, a set of logical tuples is associated, the semantics is thus obtained in two phases: (i) sending each variable to an appropriate object and (ii) extracting the logical tuples associated with the variable's instance and associating it with the variable. The logical tuple instantiations are then aggregated, restricted and/or restructured into possibly several relations which become the query result.

These notions are now formalised.

Preliminary Definitions

Formal federation instances are considered below, using the following definitions:

Let F $\in \mathcal{F}$ be a given federation instance. Then, let:

Obj_F	= the set of all objects, i.e. dbs, rel ids, col ids and tuples, appearing in federation instance F.
V_Q	= the set of all variables declared in Q.

Let there be respectively d, r, c and t database, relation, column and tuple variables declared in the FROM clause of Q (d, r, c and $t \geq 0$ and *at least one* is ≥ 1).

Then, let $D \in \{D_1, D_2, ..., D_d\}$, one of the database variables
$R \in \{R_1, R_2, ..., R_r\}$, one of the relation variables
$C \in \{C_1, C_2, ..., C_c\}$, one of the column variables
$T \in \{T_1, T_2, ..., T_t\}$, one of the tuple variables

Let us also define the following sets of dimensions by looking at the SELECT clause of Q:

Let $Simp = \{simp_1, simp_2, ..., simp_s\}$ be the set of selected non-aggregated and aggregated dimensions in `simple_col_list` and `agg_simple_col_list`.

Let $Comp = \{comp_1, comp_2, ..., comp_q\}$ be the set of selected non-aggregated and aggregated dimensions in `complex_col_list` and `agg_complex_col_list`.

Let $Colcrit = \{crit_1, crit_2, ..., crit_p\}$ be the set of dimensions in criteria position in `complex_col_list` and `agg_complex_col_list` (i.e. in inner FOR sub-clauses).

Let $\mathcal{R}elcrit = \{relcrit_1, relcrit_2, ..., relcrit_k\}$ be the set of dimensions in relation criteria position in `outer_for_crit_list`.

Let $\mathcal{N}on\mathcal{A}ggdim = \{dim_1, dim_2, ..., dim_n\}$ be the set of non-aggregated dimensions in simple_col_list and complex_col_list.

Let $\mathcal{A}ggexpdim = \{AGG_1(aggdim_1), AGG_2(aggdim_2), ..., AGG_a(aggdim_a)\}$ be the set of aggregate expressions in agg_simple_col_list and agg_complex_col_list.

Let $\mathcal{A}ggdim = \{aggdim_1, aggdim_2, ..., aggdim_a\}$ be the set of aggregated dimensions in agg_simple_col_list and agg_complex_col_list.

In the following, a vector notation will be used as shorthand to simplify some enumerations, as follows:

- $T.\vec{simp}(\texttt{rel}) = T.simp_1, T.simp_2, ..., T.simp_u$, where the *simps* are all the simple columns of rel;
- $C.\vec{crit}(\texttt{col}) = C.crit_1, C.crit_2, ..., C.crit_v$

and

- $val(\texttt{col}, \vec{crit}(\texttt{col})) = val(\texttt{col}, crit_1), val(\texttt{col}, crit_2), ..., val(\texttt{col}, crit_v)$, where the *crits* are the criteria of column col;
- $R.\vec{crit}(\texttt{rel}) = R.crit_1, R.crit_2, ..., R.crit_w$

and

- $val(\texttt{rel}, \vec{crit}(\texttt{rel})) = val(\texttt{rel}, crit_1), val(\texttt{rel}, crit_2), ..., val(\texttt{rel}, crit_w)$, where the *crits* are the criteria of relation rel;
- $(\vec{Comp}, Colcrit) = (comp_1, Colcrit), (comp_2, Colcrit), ..., (comp_q, Colcrit)$, the enumeration of the complex column schema that should appear in the query output.

As an illustration of the above, for query Q13, we would have:

$\mathcal{S}imp = \{\text{T.Date}\}$

$\mathcal{C}omp = \{\text{T.C}\}$

$\mathcal{C}olcrit = \{\text{C.Measure}\}$

$\mathcal{R}elcrit = \{\}$

$\mathcal{N}on\mathcal{A}ggdim = \{\text{T.Date}\}$

$\mathcal{A}ggexpdim = \{\texttt{SUM(T.C)}\}$

$\mathcal{A}ggdim = \{\text{T.C}\}$

$\text{T.}\vec{simp}(\texttt{bse :: prices}) = \text{T.Date}$

$\text{C.}\vec{crit}(\texttt{open\_ibm}) = \texttt{C.Ticker}, \text{C.Measure}$

$val(\texttt{open\_ibm}, \vec{crit}(\texttt{open\_ibm})) = val(\texttt{open\_ibm}, \texttt{Ticker}), val(\texttt{open\_ibm}, \texttt{Measure})$

$(\vec{Comp}, Colcrit) = (\text{T.C}, \{\text{C.Measure}\})$

Instantiation of Variables

Definition 4.3.1 *[Instantiation]*
An instantiation $\imath$ is a function $\imath : V_Q \rightarrow Obj_F$ such that $\imath$ maps each variable in V_Q to an object in the appropriate range, i.e. each db var to a db, rel var to a rel id, col var to a col id, and tuple var to a tuple (see the definition of a federation instance). Instantiations $\imath$ are also extended such that:

(i) for each literal constant `c`, $\imath(\texttt{c}) \equiv \texttt{c}$;

(ii) whenever `T` is a tuple variable declared as `db::rel T` and `col` is one of the simple columns of $\imath(\texttt{rel})$, $\imath(\texttt{T.col})$ appears in column `col` in relation $\imath(\texttt{rel})$, and furthermore, $\imath(\texttt{T.col}) = \imath(\texttt{T})[\texttt{col}]$;[2]

(iii) whenever `C` is a col var declared as `db::rel -> C` and `crit` is one of the criteria of the columns that `C` ranges over, $\imath(\texttt{C.crit})$ is one of the values of the criterion `crit` of the columns that `C` ranges over, and furthermore, $\imath(\texttt{C.crit}) = val(\imath(\texttt{C}), \texttt{crit})$;[3]

(iv) whenever `R` is a rel var declared as `db -> R` and `crit` is one of the criteria of the relations that `R` ranges over, $\imath(\texttt{R.crit})$ is one of the values of the criterion `crit` of the relations that `R` ranges over, and furthermore $\imath(\texttt{R.crit}) = val(\imath(\texttt{R}), \texttt{crit})$;

(v) whenever `T` and `C` are tuple and col variables declared as `db::rel T` and `db::rel -> C` respectively, $\imath(\texttt{T.C})$ is one of the values appearing in column $\imath(\texttt{C})$ in the relation `rel`, and furthermore, $\imath(\texttt{T.C}) = \imath(\texttt{T})[\imath(\texttt{C})]$.

The main intuition behind the above definition is that the $\imath$ function maps each variable to an object of the appropriate type in a federation instance. Furthermore, $\imath$ also maps the properties of each object (e.g., columns, or criteria) to the instantiation.

Definition 4.3.2 *[Consistent Instantiations]*
An instantiation $\imath : V_Q \rightarrow Obj_F$ is said to be *consistent* provided it satisfies the following conditions.

- Whenever `R` is a rel var declared as `db -> R`, $\imath(\texttt{R})$ is mapped to a rel id corresponding to the database $\imath(\texttt{db})$, i.e. $\imath(\texttt{R}) \in rel(\imath(\texttt{db}))$.

[2] As usual, t[col] denotes the restriction of tuple t to column col.
[3] See Def. 2.2 for the definition of the *val* function.

- Whenever `T` is a tuple variable declared as `db::rel T`, $\imath$(`T`) appears in relation $\imath$(`rel`) in database $\imath$(`db`), i.e. $\imath$(`T`) $\in$ *tup*($\imath$(`rel`)).
- Whenever `C` is a column variable declared as `db::rel -> C`, $\imath$(`C`) is mapped to a column id corresponding to relation $\imath$(`rel`) in database $\imath$(`db`), i.e. $\imath$(`C`) $\in$ *col*($\imath$(`rel`)).

Definition 4.3.3 *[Valid Instantiations]*
An instantiation $\imath : V_Q \rightarrow Obj_F$ is said to be *valid* provided it satisfies the following conditions.

- Whenever `V` is a rel var or col var declared in the `FROM` clause, and `V ISA concept` is a condition in the `WHERE` clause, $\imath$(`V`) is mapped to a rel id or col id such that *conc*($\imath$(`V`)) = `concept`.
- Whenever `V` is a rel var or col var declared in the `FROM` clause, and `V HASA crit` is a condition in the `WHERE` clause, $\imath$(`V`) is mapped to a rel id or col id such that `crit` $\in$ *crit*($\imath$(`V`)).

The set $\mathcal{I}_{\mathrm{Q}}$ is defined as $\mathcal{I}_{\mathrm{Q}} = \{\imath \mid \imath \text{ is a consistent and valid instantiation}\}$.

E.g., for `Q13`, take the instantiation $\imath$ such that:
$\imath$(`T`) $= \langle 10|27|97, 63.67, 50.23, ..., 62.56, 48.54, ...\rangle$ and $\imath$(`C`) = open_ibm.
This instantiation is not only *consistent* but also *valid* since the column with id open_ibm has concept `Price`.

Since we only consider well-formed queries (see Section 3.1.3), we thus have that $\forall$ rel var R and col var C:

$$\forall \imath, \jmath \in \mathcal{I}_{\mathrm{Q}} : crit(\imath(R)) = crit(\jmath(R))$$
$$\forall \imath, \jmath \in \mathcal{I}_{\mathrm{Q}} : crit(\imath(C)) = crit(\jmath(C))$$
$$\forall \imath, \jmath \in \mathcal{I}_{\mathrm{Q}} : col(\imath(R)) = col(\jmath(R))$$

Logical Tuples

After having obtained an instantiation for the variables in the query, we must obtain a *logical tuple* corresponding to this instantiation, in the following way.

Let the set $\mathcal{L}og\mathcal{T}up_Q$ be the set of all possible tuples over the set of logical dimensions associated with query Q.

Let $C_1, C_2, ..., C_{s_i}$ be the s_i column variables having the same db and rel component as T_i in their declarations ($s_i \geq 0$).

Definition 4.3.4 *[Logical Instantiation]*
We define a *logical instantiation* as a function $\imath_{logic} : \mathcal{I}_{\mathrm{Q}} \rightarrow \mathcal{L}ogTup_Q$ which, given an instantiation $\imath$ for the variables in Q, produces a tuple over an appropriate set of dimensions.

The function $\imath_{logic}(\imath)$ produces a logical tuple as follows:

$$\imath_{logic}(\imath) = \bigotimes_{1 \leq i \leq d} (\imath(D_i)) \bigotimes_{1 \leq i \leq t,\ 1 \leq j \leq u_i} (\imath(T_i)[simp_{i,j}]) \bigotimes_{1 \leq i \leq t,\ 1 \leq j \leq l_i} (\imath(T_i)[\imath(C_{T_{i,j}})])$$
$$\bigotimes_{1 \leq i \leq c\ 1 \leq j \leq v_i} (val(\imath(C_i), crit_{i,j})) \bigotimes_{1 \leq i \leq r\ 1 \leq j \leq w_i} (val(\imath(R_i), crit_{i,j}))$$

where d, r, c, and t are respectively the number of database, relation, column and tuple variables in the query, u_i is the number of simple columns associated with variable T_i, l_i is the number of column variables $C_{T_{i,j}}$ having the same db and rel component as T_i in their declarations, v_i is the number of criteria associated with column variable C_i and w_i is the number of criteria associated with relation variable R_i.

If there are no database, relation and column variables in the query, the function can be simplified and becomes:

$$\imath_{logic}(\imath) = \bigotimes_{1 \leq i \leq t,\ 1 \leq j \leq u_i} \imath(T_i)[simp_{i,j}]$$

which corresponds to the SQL case.

Using our earlier example instantiation, one of the logical tuples corresponding to instantiation $\imath$ is $\imath_{logic}(\imath) = \langle 10|27|97, open, ibm, 63.67\rangle$, a logical tuple over the logical dimensions {T.Date, C.Measure, C.Ticker, T.C}.

Satisfying Logical Instantiations

The conditions other_where_conds[4] in the WHERE clause induce a boolean function, denoted $sat(\imath_{logic}(\imath), Q)$, on the set of all logical tuple instantiations, reflecting whether the conditions are satisfied by an instantiation. This is defined in the obvious manner. Let $\mathcal{I}_{logic_Q} = \{\imath_{logic}(\imath) \mid \imath \in \mathcal{I}_{\mathrm{Q}} \wedge \imath_{logic}(\imath)$ is a logical tuple for which $sat(\imath_{logic}(\imath), Q) = true\}$ denote the set of logical tuples satisfying the conditions in the WHERE clause.

The set $\mathcal{I}_{logic_{Q13}}$ is shown in figure 10.

[4]Recall that other_where_conds are similar to the regular conditions allowed in the WHERE clause of SQL queries.

T.Date	C.Measure	C.Ticker	T.C
10|27|97	open	ibm	63.67
10|27|97	open	ms	50.23
...	...	...	...
10|27|97	close	ibm	62.56
10|27|97	close	ms	48.54
...	...	...	...
11|01|97	open	ibm	65.23
11|01|97	open	ms	44.60
...	...	...	...
11|01|97	close	ibm	63.05
11|01|97	close	ms	46.17
...	...	...	...

Figure 10: The set of logical instantiations $\mathcal{I}_{logic_{\texttt{Q13}}}$ corresponding to query Q13

Discarding Extraneous Dimensions

Once the satisfying logical instantiations have been obtained some dimensions may have to be discarded before structuring the data according to the proper output structure.

Let the set of dimensions $\texttt{SELECT}_Q$ be those dimensions mentioned in the SELECT clause. We define a function $disc_Q$: $\mathcal{L}ogTup_Q \rightarrow \texttt{SELECT}_Q$ as follows:

Given a tuple $\lambda \in \mathcal{I}_{logic_Q}$, then

$$disc_Q(\lambda) = \bigotimes_{D \in \texttt{SELECT}_Q} \lambda[D]$$

The set of all restricted satisfying logical tuples is then defined to be:

$$\mathcal{I}_{logic\_rest_Q} = \{disc_Q(\lambda) \mid \lambda \in \mathcal{I}_{logic_Q}\}$$

Aggregation

Similarly to the SQL case, to account for aggregation, we need the following extension. We define a relation $\sim$ on the logical tuples.

Definition 4.3.1 For λ, κ two logical tuples $\in \mathcal{I}_{logic_Q}$, $\lambda \sim \kappa$ iff $\forall$ *dimensions* $D \in$ groupby_list, $\lambda[D] = \kappa[D]$. It is straightforward to see that $\sim$ is an equivalence relation on $\mathcal{I}_{logic_Q}$. This definition essentially says that two logical tuples (satisfying the conditions in the WHERE clause) are $\sim$-equivalent provided they agree on all the GROUP BY dimensions.

Let $\mathcal{T}_{\mathrm{agg}_Q}$ denote the set of tuples over the dimensions in groupby_list and those in aggregate expressions in agg_simple_col_list, agg_complex_col_list and the HAVING clause. We define a function $agg_Q : \mathcal{I}_{logic_Q} \rightarrow \mathcal{T}_{\mathrm{agg}_Q}$ as follows.

$$agg_Q(\lambda) = \bigotimes_{\text{“}D'\text{”} \in \texttt{groupby\_list}} \lambda[D'] \bigotimes_{\text{“}agg_D(D)\text{”} \in \mathcal{A}ggexpdim \,\cup\, \texttt{having\_conds}} agg_D([\kappa[D] \mid \kappa \in \mathcal{I}_{logic_Q}, \kappa \sim \lambda])$$

For a given logical tuple λ, agg_Q considers all logical tuples equivalent to λ, and, for each aggregate operation, say agg_D, indicated on the dimension D in the agg_simple_col_list, in the agg_complex_col_list or in the HAVING clause, it performs the operation agg_D on the *multiset* of values associated with this dimension by any logical tuple equivalent to λ. We use $[\ldots]$ instead of $\{\ldots\}$ to denote multisets for aggregation. The function agg_Q also appends to the aggregated values the values of the dimensions grouped by.

Note that if there are no complex_col_list, agg_complex_col_list and outer_for_crit_list, $agg_Q(\lambda)$ reduces to:

$$agg_Q(\lambda) = \bigotimes_{\text{“}D'\text{”} \in \texttt{groupby\_list}} \lambda[D'] \bigotimes_{\text{“}agg_D(D)\text{”} \in \texttt{agg\_simple\_col\_list} \,\cup\, \texttt{having\_conds}} agg_D([\kappa[D] \mid \kappa \in \mathcal{I}_{logic_Q}, \kappa \sim \lambda]$$

which is equivalent to the SQL case.

We define the set of logical tuples $\mathcal{I}_{agglogic_Q}(\lambda)$ as:

$\mathcal{I}_{agglogic_Q} = \{agg_Q(\lambda) \mid \lambda \in \mathcal{I}_{logic_Q} \wedge agg_Q(\lambda)$ satisfies the conditions in having_list$\}$.

As an illustration, the set $\mathcal{I}_{agglogic_{\mathrm{Q13}}}$ is shown in figure 11.

Also in the case of aggregation, once the satisfying aggregated tuples have been obtained some dimensions may have to be discarded before structuring the data according to the proper output structure.

Let the set of dimensions $\mathcal{T}_{\mathrm{SELECT}_Q}$ be those dimensions mentioned in the SELECT clause. We define a function $disc_{agg_Q}$: $\mathcal{T}_{\mathrm{agg}_Q} \rightarrow \mathcal{T}_{\mathrm{SELECT}_Q}$ as follows:

T.Date	C.Measure	T.C
10|27|97	open	6367575
10|27|97	close	6368342
10|27|97	low	6360329
...	...	...
11|01|97	open	6367111
11|01|97	close	6368340
11|01|97	low	6366500
...	...	...

Figure 11: The set of logical tuples $\mathcal{I}_{agglogic_{\texttt{Q13}}}$ corresponding to query Q13

Given a tuple $\tau \in \mathcal{I}_{agglogic_Q}$, then

$$disc_{agg_Q}(\tau) = \bigotimes_{D \,\in\, \mathcal{A}ggdim} (\tau[agg_D(D)]) \quad \bigotimes_{D' \,\in\, \mathcal{N}on\mathcal{A}ggdim \,\cup\, \mathcal{R}elcrit} (\tau[D'])$$

The set of all restricted aggregated tuples is then defined to be:

$$\mathcal{I}_{agglogic\_rest_Q} = \{disc_{agg_Q}(\tau) \mid \tau \in \mathcal{I}_{agglogic_Q}\}$$

Structuring of Data Along the Physical Dimensions

The tuples in $\mathcal{I}_{logic\_rest_Q}$ (or $\mathcal{I}_{agglogic\_rest_Q}$ for aggregation queries) contain all the data necessary to answer the query. But that data may need to be structured in a set of relations each having perhaps a number of complex columns.

We need to define another equivalence relation:

Definition 4.3.2 Let ν, μ *be two tuples* $\in \mathcal{I}_{logic\_rest_Q}$. We define $\nu \equiv \mu$, provided $\nu[D] = \mu[D]$ $\forall$ D in `outer_for_crit_list`. Clearly $\equiv$ is an equivalence relation.

We can define the equivalence relation $\equiv$ on tuples belonging to $\mathcal{I}_{agglogic\_rest_Q}$ in a similar fashion. In the sequel, we will assume that $\equiv$ is defined on both the sets of tuples $\mathcal{I}_{logic\_rest_Q}$ and $\mathcal{I}_{agglogic\_rest_Q}$.

We see that the equivalence relation $\equiv$ partitions the logical tuples (or aggregated logical tuples) in sets of tuples agreeing on all dimensions to be placed in relation positions.

The logical tuples $\equiv$-equivalent to (aggregated) logical tuple ν will thus contribute to a relation, $struct_Q(\nu)$ with schema $(output, \mathcal{R}elcrit)(Simp, (\vec{Comp}, Colcrit))$, as follows.

Let $\mathcal{S}_o$ be the set of column instance corresponding to the schema of the tuples in $\mathcal{I}_{logic\_rest_Q}$ (or $\in \mathcal{I}_{agglogic\_rest_Q}$ in the case of an aggregate query). For each logical tuple $\mu \in \mathcal{I}_{logic\_rest_Q}$ (or in $\mathcal{I}_{agglogic\_rest_Q}$ for the aggregation case), such that $\mu \equiv \nu$, $struct_Q(\nu)$ contains a tuple t, defined as follows. Let $A \in \mathcal{S}_o$. Then

$$t[A] = \begin{cases} \mu[D], & \text{whenever } A = D \in \texttt{simple\_col\_list} \\ \mu[agg_D(D)], & \text{whenever } A = D \in \texttt{agg\_simple\_col\_list} \\ \mu[D], & \text{whenever } A = (D, <\mu[crit_1], \mu[crit_2], ..., \mu[crit_p]>), \\ & \qquad D \in \texttt{complex\_col\_list} \\ \mu[agg_D(D)], & \text{whenever } A = (D, <\mu[crit_1], \mu[crit_2], ..., \mu[crit_p]>), \\ & \qquad D \in \texttt{agg\_complex\_col\_list} \\ null, & \text{otherwise.} \end{cases}$$

Figure 12(i) shows $struct_{\text{Q13}}(\nu)$ for query (Q13), where ν is any logical tuple for Q13 (all of them are $\equiv$-equivalent).

T.Date	(SUM(T.C), <open>)	(SUM(T.C), <close>)	(SUM(T.C), <low>)	...
10|27|97	6367575	null	null	...
10|27|97	null	6368342	null	...
10|27|97	null	null	6360329	...
...	...	...	...	...
11|01|97	6367111	null	null	...
11|01|97	null	6368340	null	...
11|01|97	null	null	6366500	...
...	...	...	...	...

(i)

T.Date	(SUM(T.C), <open>)	(SUM(T.C), <close>)	(SUM(T.C), <low>)	...
10|27|97	6367575	6368342	6360329	...
11|01|97	6367111	6368340	6366500	...
...	...	...	...	...

(ii)

Figure 12: (i) The relation $struct_{\text{Q13}}(\nu)$ and (ii) the final result after merging

Finally, the tuples in $struct_Q(\nu)$ are merged in the following manner:

Let DOM denote the union of all values $\in \mathcal{V}al_F$, together with the null value, *null*. Define a partial order on DOM, by setting $null \leq v$, $\forall v \in$ DOM. In particular, note that any two distinct non-null values are incomparable. The *least upper bound lub* of two values in DOM is defined in the obvious way.

$$lub(u,v) = \begin{cases} u, & \text{if } v \leq u \\ v, & \text{if } u \leq v \\ undefined, & \text{otherwise.} \end{cases}$$

We now have the following

Definition 4.3.3 Two tuples t_1, t_2 over a relation scheme $R = \{C_1, \ldots, C_n\}$ of column instances C_i are *mergeable* provided for each $i = 1, \ldots, n$, either $t_1[C_i] = t_2[C_i]$, or at least one of $t_1[C_i]$ or $t_2[C_i]$ is a null.

Definition 4.3.4 Let two tuples t_1, t_2 over a relation scheme $R = \{C_1, \ldots, C_n\}$ be mergeable. Then their *merge*, denoted $t = t_1 \odot t_2$, is defined as $t[C_i] = lub(t_1[C_i], t_2[C_i])$, $i = 1, \ldots, n$. We extend this definition to an arbitrary set of mergeable tuples. A set of tuples is said to be mergeable if each pair of tuples in the set are mergeable. The merge of a set $\{t_1, t_2, \ldots, t_n\}$ of mergeable tuples is defined as $\odot\{t_1, t_2, \ldots, t_n\} = t_1 \odot (t_2 \odot (\ldots \odot (t_{n-1} \odot t_n) \ldots))$ Clearly, the operator $\odot$ is commutative and associative.

It should be noted that mergeability is *not* an equivalence relation. In general, three tuples t_1, t_2 and t_3 can be such that t_2 is mergeable with t_1 or t_3 but t_3 is not mergeable with t_1. For example, the set of tuples $t_1 = \langle 10|27|97, 6367575, null, null\rangle$, $t_2 = \langle 10|27|97, null, 6368342, null\rangle$ and $t_3 = \langle 10|27|97, 6367222, null, null\rangle$ present this behaviour. Thus one can partition this set of tuples into sets of mergeable tuples in several ways. The possibilities are $P_1 = \{\{t_1, t_2\}, \{t_3\}\}$, $P_2 = \{\{t_1\}, \{t_2, t_3\}\}$ or even the trivial partition $P_3 = \{\{t_1\}, \{t_2\}, \{t_3\}\}$

It will be convenient below to extend the operator $\odot$ to any relation containing an arbitrary (*i.e. not* necessarily mergeable) set of tuples.

Let r be any relation over the scheme R. Using the mergeability property, we can partition the relation into sets of mergeable tuples. Let $\mathcal{MP}(r)$ be the set of all such partitions of mergeable tuples.

Now recall the notion of *refinement* of partitions. We say that partition P_1 is a refinement of partition P_2 $\Leftrightarrow$ $\forall\ cell\ p_i \in P_1\ \exists\ some\ cell\ p_j \in P_2\ s.t.\ p_i \subseteq p_j$. The notion of refinement defines a partial order over partitions such that *a maximal partition with respect to refinement* is one which is not a refinement of any other partition.

Using the preceding example of partitions of mergeable tuples, we have that partition P_3 is a refinement of both P_1 and P_2, while the latter two partitions are each maximal.

Using the notion of refinement of partitions, we can compare the elements of $\mathcal{MP}(r)$. Note that in general there may be several maximal elements of $\mathcal{MP}(r)$, as seen in the previous example.

We now define the merge of a relation as:

Definition 4.3.5 Let r be any relation over the scheme R. Let Π be any maximal element of $\mathcal{MP}(r)$. Then the *merge* of r, denoted $\bigodot r$, is defined as follows.

$\bigodot r = \{t \mid \{t_1, \ldots, t_m\} \in \Pi \wedge t = \bigodot\{t_1, \ldots, t_m\}\}$.

Note that $\bigodot\{t_1, \ldots, t_m\}$ is as defined in Definition 4.3.4.

Notice that $\bigodot r$ is defined based on *any* maximal element of $\mathcal{MP}(r)$. Thus, choosing different maximal elements of $\mathcal{MP}(r)$ can give rise to different values for $\bigodot r$. We shall see below that the difference resulting from this is irrelevant as far as the information content is concerned.

The output relation produced by logical tuples $\equiv$-equivalent to $\nu \in \mathcal{I}_{logic\_rest_Q}$ (or $\mathcal{I}_{agglogic\_rest_Q}$ for an aggregation query) is given by $\bigodot struct_Q(\nu)$ and has relation label $(output, < \nu[relcrit_1],\ \nu[relcrit_2],\ \ldots,\ \nu[relcrit_k] >)$.

Since the $\bigodot$ operator is defined as using any maximal mergeable partition of $struct_Q(\nu)$, it is important to note that every application of the $\bigodot$ operator results in sets of tuples that are *equivalent with respect to the underlying logical tuples.*

For example, in the case of the three tuples t_1, t_2 and t_3, their merge could be either:

$\{\langle 10|27|97, 6367575, 6368342, null\rangle, \langle 10|27|97, 6367222, null, null\rangle\}$

or

$\{\langle 10|27|97, 6367575, null, null\rangle, \langle 10|27|97, 6367222, 6368342, null\rangle\}$

according respectively to partitions P_1 and P_2.

Finally, we can define the semantics of an nD-SQL query as follows. $Q(F) = \{\bigodot struct_Q(\nu) \mid \nu \in \mathcal{I}_{logic\_rest_Q}$ *(or* $\mathcal{I}_{agglogic\_rest_Q}$ *for an aggregation query)*$\}$ This is a set of relations if $\mathcal{R}elcrit$ is not empty. Note that if the set $Colcrit$ is empty (i.e. if there is no inner FOR clause in the query), the tuples of $struct_Q$ do not need to be merged. If the set $\mathcal{R}elcrit$ was also empty (i.e. if there were no outer FOR sub-clauses either), then the query result is simply $\mathcal{I}_{logic\_rest_Q}$ (or $\mathcal{I}_{agglogic\_rest_Q}$ for an aggregation query).

As an example, the final output produced by query (Q13) is relation output(T.Date, (T.C,<open>), (T.C,<close>), ...) as shown in Figure 12(ii).

Chapter 5

Query Processing

This chapter discusses the efficient implementation of the nD-SQL language. The first section (5.1) deals with queries that do not involve dimension variables. The processing of queries involving dimension variables is covered in Section 5.2 In this chapter, all queries are assumed to be *explicit* (i.e. all abbreviations have been expanded). Section 5.1.1 presents the Restructuring Relational Algebra (RRA), used for query processing. Section 5.1.2 then presents an algorithm to obtain an RRA expression equivalent to a given nD-SQL query. Section 5.1.3 will discuss optimisation.

5.1 Processing of Queries that Do Not Involve Dimension Variables

In order to efficiently process nD-SQL queries, an extension to classical Relational Algebra (RA) is proposed. Restructuring Relational Algebra (RRA) extends classical RA with restructuring operators. Thus, to process nD-SQL queries, a translation of the query will be made into an equivalent RRA expression, just like SQL queries are translated into RA expressions. This will permit the processing engine to take advantage of the properties of the RRA operators to optimise the expressions. It is also possible to take advantage of downward compatibility of RRA with RA to push some of the processing to remote databases. The query processing architecture is illustrated in Figure 13. Its highlights are that it is non-intrusive, requiring minimal extensions to existing technology, for deployment on top of existing SQL systems. Before going into details about processing, the RRA is defined next.

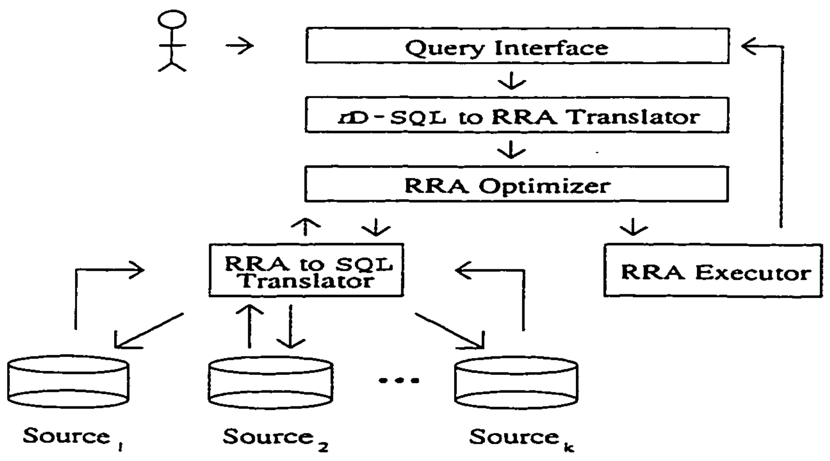

Figure 13: nD-SQL Server Architecture

5.1.1 Restructuring Relational Algebra

RRA consists of the classical RA operators (that are slightly extended), together with new restructuring operators. These address the issues arising from: (i) complex relations and columns; (ii) restructuring with a dynamic input and/or output schema. Recall that in the model (see Chapter 2), simple columns of relations are denoted as in the classical relational model, while complex columns are of the form (concept, $t_{\texttt{criteria}}$). The operators of RRA are thus: σ, Π, $\times$, ρ, ADD_COL, REM_COL, ADD_REL and AGG where the latter supports any of the usual aggregations.

First, the new operators are defined, then the extensions to the classical operators are explained.

Definition 5.1.1 (Add Criteria to Columns)
The operation $\text{ADD\_COL}_{critList \to concList}(\texttt{rel})$, *critList and concList being sets of concepts, applied to a relation with name* `rel`, *has the following effect. Let r be any instance of the relation name* `rel` *in the database. Then, the operation produces an output relation r' with the same concept as r, satisfying the following conditions.*

- *The column labels of r' are* $cols(r') = (cols(r) - \{A \mid A \text{ is a simple column of } r \text{ with concept in } critList\} - \{B \mid B \text{ is a column of } r \text{ with concept in } concList\}) \cup \{(C, t_C \otimes t_{critList}) \mid (C, t_C) \text{ is a column of } r \text{ with concept in } concList \wedge \exists t \in r : t[critList] = t_{critList}\}$. $\otimes$ *denotes the concatenation of tuples.*

- *The instance of r' consists of a set of tuples over $cols(r')$, defined as* $inst(r') = \{t \mid \forall (C, t_C \otimes t_{critList}) \in cols(r') - cols(r) : \exists s \in r : \forall A \in cols(r) \cap cols(r') : t[A] = s[A] \wedge t_{critList} = s[critList] \wedge t[(C, t_C \otimes t_{critList})] = s[(C, t_C)]\}$.

It should be noted that the column (C, t_C) could be a simple or complex column, in the above definition. As an illustration of the above operator, the expression $\text{ADD\_COL}_{\text{Measure} \rightarrow \text{Price}}(\texttt{nyse :: prices})$ would produce a relation with column labels similar to those of `tse::quotes` of Figure 1, and contents equivalent to those of `nyse::prices`. The resulting table, call it ny2t, is shown in Figure 14.

Ticker	Date	open	close	...
ibm	10|27|97	63.67	62.56	...
...	...	...	...	...
ms	11|01|97	44.60	46.17	...

Figure 14: `ny2t::prices`

An algorithm that implements the ADD_COL operator is given in figure 15.

```
INPUT: Table $\mathcal{T}_i$ with schema $\mathcal{R}_i$
        Set $\mathcal{C}$ of criteria to add
        Set $\mathcal{S}$ of concepts to add criteria to

OUTPUT: Table $\mathcal{T}_o$ with schema $\mathcal{R}_o$


let $\mathcal{R}_S$ be the set of complex columns from $\mathcal{R}_i$ with their concept in $\mathcal{S}$;
$\mathcal{R}_o = \mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)$;
for each tuple $t_j$ in $\mathcal{T}_i$ do
    let $t_j[\mathcal{C}]$ be the restriction of $t_j$ over the criteria in $\mathcal{C}$
    for each complex column $(s_i, t_{i,j})$ in $\mathcal{R}_S$ do
        $\mathcal{R}_o = \mathcal{R}_o \cup \{(s_i, t_{i,j} \otimes t_j[\mathcal{C}])\}$;
    end for
    let $t_j[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)]$ be the restriction of $t_j$ over the columns of $\mathcal{R}_i$ neither in $\mathcal{C}$ nor in $\mathcal{R}_S$
    if there does not exist a Merge Class for $t_j[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)]$ then
      create the Merge Class;
      add $t_j$ to the Merge Class;
    else
      if in some Merge Class for $t_j[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)]$ there is no tuple with the same set of values of $t_j[\mathcal{C}]$ then
        add $t_j$ to that Merge Class;
      else
        create another Merge Class for $t_j[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)]$;
        add $t_j$ to that Merge Class;
      end if
    end if
end for

for each Merge Class do
    initialise output tuple $t_o$ for $\mathcal{T}_o$ to all nulls;
    let $t_k$ be any tuple in the Merge Class
    $t_o[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)] = t_k[\mathcal{R}_i - (\mathcal{C} \cup \mathcal{R}_S)]$;
    for each tuple $t_j$ in the Merge Class do
        for each $(s_i, t_{i,j}) \in \mathcal{R}_S$ do
            $t_o[(s_i, t_{i,j} \otimes t_o[\mathcal{C}])] = t_j[(s_i, t_{i,j})]$;
        end for
    end for
end for
```

Figure 15: Algorithm for RRA operator ADD_COL

Definition 5.1.2 (Remove Criteria from Columns) *The operation*

$\text{REM\_COL}_{\text{critList}\leftarrow\text{concList}}(\texttt{rel})$. *critList being a list of criteria and concList a list of concepts, applied to a relation with name* `rel`, *has the following effect. Let* r *be any instance of the relation name* `rel` *in the database. Then, corresponding to each such relation* r, *the operation produces an output relation* r', *with the same concept as* r, *satisfying the following conditions.*

- *The column labels of* r' *are* $cols(r') = \{A \mid A \text{ is a simple column in } cols(r)\} \cup \{(B, t_C)[C - critList] \mid (B, t_C) \text{ is a complex column in } cols(r) \text{ with criteria set } C\} \cup (critList \cap C)$. *Here* $cols(r)$ *is the set of column labels of* r.
- *The instance of* r' *consists of a set of tuples over* $cols(r')$, *defined as* $inst(r') = \{t \mid \exists s \in r : \exists \text{ a criteria-tuple } t_C : (\forall \text{ simple column } A \in cols(r) : t[A] = s[A]) \wedge (\forall \text{ complex column } (C, t_C) \in cols(r) : t[(C, t_C[C - critList])] = s[(C, t_C)]) \wedge t[critList \cap C] = t_C[critList \cap C]\}$.

As an illustration, the expression $\text{REM\_COL}_{\text{Measure}\leftarrow\text{Price}}(\texttt{ny2t :: prices})$, applied to the relation `ny2t::prices` of Figure 14, exactly yields the relation `nyse::prices` of Figure 1.

An algorithm that implements the REM_COL operator is given in figure 16.

```
INPUT: Table $\mathcal{T}_i$ with schema $\mathcal{R}_i$
       Set $C$ of criteria to remove

OUTPUT: Table $\mathcal{T}_o$ with schema $\mathcal{R}_o$


$\mathcal{R}_o = \mathcal{R}_i \cup C$;
for each column $r_i$ in $\mathcal{R}_i$ do
    if $r_i$ is an instance of a complex column then
        let $t_i$ be the criteria_tuple of $r_i$ and $C_{all}$ be the criteria_set of $r_i$
        $\mathcal{R}_o = \mathcal{R}_o - \{((concept(r_i), t_i)\}$;
        $\mathcal{R}_o = \mathcal{R}_o \cup \{((concept(r_i), t_i[C_{all} - C])\}$;
        store $t_i[C]$ in set criteria_tuples;
    end if
end for

for each tuple $t_j$ in $\mathcal{T}_i$
    for each criteria_tuple $t_k[C]$ in the set criteria_tuples do
        initialize output tuple $t_o$ for $\mathcal{T}_o$ to all nulls;
        let $\mathcal{R}_s$ be the set of simple columns of $\mathcal{R}_i$
        $t_o[\mathcal{R}_s] = t_j[\mathcal{R}_s]$;
        $t_o[C] = t_k[C]$;
        for each instance $r_i$ in $\mathcal{R}_i$ of a complex column do
            $t_o[(concept(r_i), t_k[C])] = t_o[r_i]$;
        end for
        if some $t_o[(concept(r_i), t_k[C])]$ is not null then
            add $t_o$ to $\mathcal{T}_o$;
        end if
    end for
end for
```

Figure 16: Algorithm for RRA operator REM_COL

Definition 5.1.3 (Add Criteria to Relations) *The operation* ADD_REL$_{\text{critList}}$(rel), *critList being a list of criteria, applied to a relation with name* rel, *has the following effect. Let r be any instance of the relation name* rel *in the database. Assume for simplicity that all criteria in critList are concepts of simple columns in r. Then, corresponding to each relation r, the operation produces multiple output relations r', with the same concept as r, and with criteria critList, that satisfy the following conditions.*

- *The column labels of every r' are $cols(r') = cols(r) - critList$*
- *There is one output relation r' corresponding to r and to each distinct critList-value, say $t_{critList}$, in r. Let the label of this relation r' be (*rel*, $t_{critList,i}$).*
- *The instance of each (*rel*, $t_{critList,i}$) consists of a set of tuples over $cols(r')$, defined as $inst(r')_i = \{t \mid \exists s \in r : (\forall A \in cols(r') : t[A] = s[A]) \wedge t_{critList,i} = s[critList]\}$.*

As an illustration, the expression ADD_REL$_{\text{Ticker}}$(ny2t :: prices) would produce multiple relations, with relation labels similar to those of mse::quotes of Figure 1, with column labels similar to the ones of those relations, and contents equivalent to those of ny2t::prices. The resulting table is shown in Figure 17.

Date	open	close	...
10|27|97	63.67	62.56	...
...	...	...	...
11|01|97	65.23	63.05	...

ny2m::ibm

Date	open	close	...
10|27|97	50.23	48.54	...
...	...	...	...
11|01|97	44.60	46.17	...

ny2m::ms

...

...

Figure 17: ny2m::prices

An algorithm that implements the ADD_REL operator is given in figure 18.

```
INPUT: Set $\mathcal{T}_i$ of tables with schema $\mathcal{R}_i$
       Set $C$ of criteria to add, $C \subseteq$ {set of simple columns in $\mathcal{R}_i$}

OUTPUT: Set $\mathcal{T}_o$ of tables with schema $\mathcal{R}_o$


initialise $\mathcal{T}_o$ to be empty;
for each table $T_i \in \mathcal{T}_i$ do
    let $t_{r_i}$ be the criteria_tuple of table $T_i$
    for each tuple $t_k$ in table $T_i$ do
        $\mathcal{T}_o = \mathcal{T}_o \cup \{(concept(T_i), t_{r_i} \oplus t_k[C])(\mathcal{R}_i - C)\}$;
        add the tuple $t_k[\mathcal{R}_i - C]$ to table $(concept(T_i), t_{r_i} \oplus t_k[C])(\mathcal{R}_i - C)$;
    end for
end for
```

Figure 18: Algorithm for RRA operator ADD_REL

It turns out the converse of ADD_REL, call it REM_REL, is not needed as an explicit operator, as its sense is built into the query processing algorithms. A more complete explanation is deferred to Section 5.1.2.

The classical RA operators are extended in the following way: we allow that parameters to these operators refer to one specific column instance of a complex column by using its label. We also also allow them to refer to the set of instances of a complex column by using the column's concept. This serves as a shorthand for enumerating every column label and applying the same (e.g. Π_{Price}(`nyse::prices`) denotes the projection of relation `nyse::quotes` on the *set* of columns having concept `Price`). This is perfectly compatible with RA since when a column is simple, this abbreviation reduces to the classical select or project.

Finally, the ρ (renaming) operator is extended in the following way: when renaming complex columns (by using as argument its concept as allowed above), complex labels are allowed (of the form used in the `SELECT` clause of an nD-`SQL` query, see point (5) of Section 3.1.1).

For simplicity in presentation, the operators have been defined as applying to single tables. Operations sometimes have to be applied to joins of tables where different columns could have the same concept. A simple way to support this is that parameter lists should accept pairs (table.concept) or (table.criteria) wherever concepts or criteria were used in the definitions (e.g. we could have $\text{AGG}_{\text{SUM(T1.Price)}, \{\text{T1.Date,T2.Date}\}}$, where `SUM(T1.Price)` is the aggregation to be performed, `1.Price` is the dimension being aggregated, and {`T1.Date, T2.Date`} is the set of group-by dimensions).

Useful Identities

An important property of the new restructuring operators is that *they preserve the logical tuples* underlying the physical data. From this, we can obtain several identities that will be useful for query processing or optimisation. These identities are based on the notions of *logical equivalence* and *n-dimensional equivalence* defined next.

Definition 5.1.4 (Logical Equivalence) We say that $Table_1$ is *logically equivalent* to $Table_2$ (denoted $Table_1 \equiv_{logic} Table_2$ iff the set of logical tuples underlying $Table_1$ is equal to the set of logical tuples underlying $Table_2$.

Definition 5.1.5 (n-Dimensional Equivalence) We say that $Table_1$ is *n-dimensionally equivalent* to $Table_2$ (denoted $Table_1 \equiv_{nD} Table_2$) iff $Table_1 \equiv_{logic} Table_2$ and

$Table_1$ and $Table_2$ have the same federated schema.

Note that when $Table_1 \equiv_{nD} Table_2$, both tables will have the same set of associated dimensions, those dimensions will be structured in the same manner along the row and column dimensions, and their set of underlying logical tuples will be the same, which means that if there are complex columns, their number of instances and the criteria tuples of those instances will be the same for both tables. Thus, $\equiv_{nD}$-equivalence can be different from equality only in terms of the merging of logical tuples.

In a similar fashion, we define $\equiv_{nD}$-equivalence of nD-SQL queries as follows:

Definition 5.1.6 (n-Dimensional Equivalence of nD-SQL queries) We say that two nD-SQL queries Q_1 and Q_2 are *n-dimensionally equivalent* (denoted $Q_1 \equiv_{nD} Q_2$) iff they give $\equiv_{nD}$-equivalent outputs when applied to $\equiv_{nD}$-equivalent inputs.

Since the restructuring operators of RRA preserve the set of logical tuples, we obtain several identities which are useful for query processing and optimisation.

First, operators of the RRA commute provided certain conditions are met. This is formalised in the following theorems.

Theorem 5.1.1 *Commutativity of operators.*

(a) $\text{ADD\_REL}_{\text{addrelcritList}}$ [$\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ $(Table)$] $\equiv_{nD}$ $\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ [$\text{ADD\_REL}_{\text{addrelcritList}}$ $(Table)$], provided the sets of domains referred to in the following pairs of parameter lists are disjoint: (i) addrelcritList, addcritList, (ii) addrelcritList, addconcList;

(b) $\text{ADD\_REL}_{\text{addrelcritList}}$ [$\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ $(Table)$] $\equiv_{nD}$ $\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ [$\text{ADD\_REL}_{\text{addrelcritList}}$ $(Table)$], provided the sets of domains referred to in the following pairs of parameter lists are disjoint: (i) addrelcritList, remcritList, (ii) addrelcritList, remconcList;

(c) $\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ [$\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ $(Table)$] $\equiv_{nD}$ $\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ [$\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ $(Table)$], provided the sets of domains referred to in the following pairs of parameter lists are disjoint: (i) addconcList, remcritList, (ii) addcritList, remconcList;

(d) Let RES-OP be either of REM_COL or ADD_COL. Then, σ_{selList} [$\text{RES-OP}_{\text{critList,concList}}$ $(Table)$] $\equiv_{nD}$ $\text{RES-OP}_{\text{critList,concList}}$ [σ_{selList} $(Table)$], provided none of the concepts and criteria in critList and concList are present in the conditions in selList;

(e) Π_{projList} [$\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ $(Table)$]
$\equiv_{nD}$ $\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ [$\Pi_{\text{projList}-\text{remcritList}}$ $(Table)$],
provided (remcritList $\cup$ remconcList) $\subseteq$ projList;

(f) $\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ [Π_{projList} $(Table)$]
$\equiv_{nD}$ $\Pi_{\text{projList}-\text{addcritList}}$ [$\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ $(Table)$],
provided (addcritList $\cup$ addconcList) $\subseteq$ projList;

(g) $\text{AGG}_{\text{aggList, groupbyList}}$ [$\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ $(Table)$]
$\equiv_{nD}$ $\text{REM\_COL}_{\text{remcritList} \leftarrow \text{remconcList}}$ [$\text{AGG}_{\text{aggList, groupbyList}-\text{remcritList}}$ $(Table)$],
provided remcritList $\subseteq$ groupbyList and every dimension aggregated in aggList is in the remconcList;

(h) $\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ [$\text{AGG}_{\text{aggList, groupbyList}}$ $(Table)$]
$\equiv_{nD}$ $\text{AGG}_{\text{aggList, groupbyList}-\text{addcritList}}$ [$\text{ADD\_COL}_{\text{addcritList} \rightarrow \text{addconcList}}$ $(Table)$],
provided addcritList $\subseteq$ groupbyList and every dimension aggregated in aggList is in the addconcList;

(i) Let RES-OP be either of REM_COL or ADD_COL. Then,
$Table_1 \times$ [$\text{RES-OP}_{\text{critList,concList}}$ $(Table_2)$]
$\equiv_{nD}$ $\text{RES-OP}_{\text{critList,concList}}$ [$Table_1 \times Table_2$],
provided the restructuring operation applies only to $Table_2$ *and that for the purpose of the restructuring operation, all columns of* $Table_1$ *are considered simple.*

PROOF:

Identities (a), (b) and (c) follow from the fact that REM_COL, ADD_COL and ADD_REL each preserve the underlying sets of logical tuples. The conditions on the parameter sets insure that one operator does not move a criteria (or concept) from (or to) the row position while the other operator was needing it in the other position.

Identity (d) follows from the fact that the logical tuples satisfying the selections are the same for both expressions, and the condition insures that the concepts and criteria involved in the restructuring are not parameters for the selection, and thus do not need to be in column positions for the selections to be performed.

Identities (e) and (f) follow from the fact that the projection does not eliminate any columns involved in the restructuring. Thus, applying the projection before of after the restructuring does not change the values involved in the restructuring. In each case, the concepts moving into criteria positions do not need to be projected on.

Identities (g) and (h) follow from the fact that whenever the values in complex columns are aggregated, the criteria set of the column act as a set of group-by dimensions. Thus, whenever the condition is met, both results will be $\equiv_{nD}$-equivalent.

Identity (i) states that by considering all columns of $Table_1$ as simple, we can apply a restructuring on the columns of $Table_2$ *even after having taken the Cartesian product of the tables.* For REM_COL, each tuple of $Table_2$ is simply broken down in many tuples *in a manner independent of the values in the columns not involved in the restructuring.* Thus, doing the scalar product before or after will change neither the number of, nor the nature of the tuples created, For ADD_COL, we observe that the operation merges logical tuples that have *identical values in all simple columns not involved in the restructuring.* Two tuples having that characteristics will necessarily produce, after a Cartesian product, pairs of tuples that still have the same property. Thus the order of execution of the two operators will produce the same result. ∎

5.1.2 Translation from nD-SQL to RRA

This section presents and explain algorithm nD-SQLtoRRA, which takes as input an nD-SQL query and produces as output an $\equiv_{nD}$-equivalent RRA expression.

Consider the generic nD-SQL query

```
SELECT      (select_object1, ..., select_objectk) FOR criteria_list
FROM        variable_declarations
WHERE       where_conditions
GROUP BY    groupby_dimensions
HAVING      having_conditions
```

where inner FOR sub-clauses may be present.

The output of algorithm nD-SQLtoRRA is a *simple* RRA expression that is equivalent to the nD-SQL query. However, the expression is in general far from optimal. The goal of the algorithm is to obtain an expression $\equiv_{nD}$-equivalent to the given nD-SQL query so that subsequently, the optimisation techniques described in Section 5.1.3 can be exploited to transform it into another $\equiv_{nD}$-equivalent, but more efficient RRA expression.

The expression resulting from nD-SQLtoRRA conforms to a basic template, T, given as follows:

$$\mathsf{T} : \mathrm{ADD\_REL}_{\mathrm{addrelList}}(\rho_{\mathrm{renamList}}(\mathrm{ADD\_COL}_{\mathrm{addList}\rightarrow\mathrm{addconcList}}(\Pi_{\mathrm{projList}}(\sigma_{\mathrm{haveList}}(\mathrm{AGG}_{\mathrm{aggList},\ \mathrm{groupbyList}}(\sigma_{\mathrm{selList}}(\mathrm{REM\_COL}_{\mathrm{remList}\leftarrow\mathrm{remconcList}}(\times_{\mathrm{prodList}}))))))))$$

The two major differences between nD-SQL query translation into an RRA expression, and the classical case of a SQL query translated into and RA expression are:

1. An nD-SQL query allows multiple types of variables, which is reflected in query processing by different types of Variable Instantiation Tables (or VITs), some for which the data needed to instantiate a variable is available in the catalog tables, some for which we need to obtain data from remote sources;
2. The restructuring capabilities of nD-SQL bring new operators into play, which translates into an additional pre-processing phase and more complex analysis of query clauses.

A very high level description of the algorithm follows.

Intuitively, it is expected that the classical SQL parts of an nD-SQL query translate into the corresponding classical RA operations (e.g. selected objects in the SELECT clause become parameters of projections, conditions in the WHERE clause become parameters to selections, etc). In addition to this, the new parts of the syntax will induce additional operations. Specifically, restructurings are derived from both (i) the new FOR sub-clauses of the SELECT clause, and (ii) the column variables declared in the FROM clause which are linked to some tuple variable through their range declaration.

The tables to which these operations will be applied are obtained from the instantiations of the variables. Note that while the information necessary to instantiate non-tuple variables is contained in the catalog tables, pulling data by querying remote sources is necessary to instantiate tuple variables. The Variable Instantiation Table containing the instantiations of a variable V_i is denoted by VIT_V_i.

The RRA subexpressions that create the VITs for the various types of nD-SQL variables can be found in Table 3. Note that to each tuple of a tuple_var is appended the name of the database and relation from which it originates. Also note that the VIT of a rel_var will contain one column per criteria of the relation (if the latter is complex). Thus, for a tuple_var declared with a complex relation in its range, the join of the tuple_var VIT with it's parent rel_var VIT gives exactly the same result as a potential REM_REL operator would. This is why there is no explicit need for an operator REM_REL in the RRA.

The algorithm nD-SQLtoRRA can be found in Figure 19. Without loss of generality, it considers an nD-SQL query where all variables are made explicit, expanding necessary abbreviations if necessary. The query is assumed to be well-typed. For simplicity in exposition, it is further assumed that the WHERE clause is a conjunction of conditions.

Algorithm *nD-SQLtoRRA* will produce an RRA expression (following template T) equivalent to the input *n*D-SQL query. The algorithm makes use of Tables 3 to 5, filling in the appropriate parameters by analysing the *n*D-SQL query. Note that in the algorithm, T denotes a tuple variable, R a relation variable, C a column variable and V either a relation or column variable. We have the following result about algorithm nD-SQLtoRRA:

Declaration	RRA subexpression for VIT
-> D	$\mathtt{VIT\_db} = \Pi_{db}(\mathtt{dbscheme})$
db -> R	$\rho_{(value,criteria)}\{\mathrm{ADD\_COL}_{\mathrm{criteria}\rightarrow\mathrm{value}}(\Pi_{db,rel\_label,criteria,value}[\mathtt{dbscheme} \bowtie_{relid=relid} \mathtt{relschemes} \bowtie_{relid=id} \mathtt{criteria}])\}.$
db::rel C	$\rho_{(value,criteria)}\{\mathrm{ADD\_COL}_{\mathrm{criteria}\rightarrow\mathrm{value}}(\Pi_{db,rel\_label,attr\_label,criteria,value}[\mathtt{dbscheme} \bowtie_{relid=relid} \mathtt{relschemes} \bowtie_{attrid=id} \mathtt{criteria}])\}.$
db::rel T	$\bigcup_{db_i} [\ \{'db_i'\} \times \bigcup_{rel_i} (\ \{'rel_i'\} \times rel_i\ )\ ]$ where $db_i \in$ {range of the db component of T's declaration} and $rel_i \in$ {range of the rel component of T's declaration} and we assign the concept db to the column with values 'db_i' and the concept rel_label to the column with values 'rel_i'

We add to the expressions for rel_vars and col_vars appropriate selections corresponding to the ISA and HASA constraints present in the WHERE clause, and to each constant present in the ranges of variable declarations in the FROM clause. For example, for a col_var C declared with the range d::rel -> and the constraint C ISA Price, the VIT will be created by adding inside the projection the selections db = 'd' AND rel_label = 'rel' AND attr_concept = Price.

Also, for relation and column variables, whenever the db component of the range is a variable, the database variable should be instantiated first and a selection added to the VIT subexpression of the rel of col var of the type db = 'db_1' OR db = 'db_2' OR ... OR db = 'db_n' where the db_i's are the databases the db var ranges over.

Table 3: RRA subexpressions for VITs of variables declared in the FROM clause

Theorem 5.1.2

The RRA expression obtained by executing algorithm nD-SQLtoRRA for an nD-SQL *query* Q *is* $\equiv_{nD}$*-equivalent to the query* Q.

```
Algorithm nD-SQLtoRRA

// definitions

$\mathcal{R}el_D$ = {$R_i$ | $R_i$ is a rel_var declared in the FROM
        clause ∧ $R_i$ has D as db component in
        its declaration}
$Par_C$ = {$T_i$ | $T_i$ is a tuple_var declared in the FROM
        clause ∧ $T_i$ has the same db and rel component
        than C in its declaration}
$Child_R$ = {$T_i$ | $T_i$ is a tuple_var declared in the FROM
        clause ∧ R is the rel component in
        $T_i$'s declaration}
$concept(V)$ = concept associated with col_var
        (or rel_var) V
$criterion(V)$ = set of criteria associated with col_var
        (or rel_var) V
$paramList^{VIT\_V}$ = {$domain_i$ |
        VIT_V.$domain_i$ ∈ paramList}

Let ADD($domain$) =

  VIT_D.(db)                 if domain is o.t.f. D
  VIT_T.(attribute)          if domain is o.t.f. T.(attribute)
  VIT_T.concept(C)           if domain is o.t.f. T.C
  VIT_R.(criterion)          if domain is o.t.f. R.(criterion)
  VIT_C.(criterion)          if domain is o.t.f. C.(criterion)
                             ∧ $||Par_C|| = 0$
  VIT_$T_C$.(criterion)      if domain is o.t.f. C.(criterion)
                             ∧ $||Par_C|| > 0$

// pre-processing

for each declared col_var C do
    generate set $Par_C$;
    identify concept(C);
    generate set criteria(C);
    if $||Par_C|| > 0$ then
        add VIT_$T_i$.$crit_j$ to remList
            and VIT_$T_i$.concept(C) to remconcList,
            ∀ $T_i \in Par_C$, ∀ $crit_j \in criteria(C)$;
        if $||Par_C|| > 1$ then
            add VIT_$T_i$.$crit_j$ = VIT_$T_k$.$crit_j$ to selList,
                ∀ $T_i, T_k \in Par_C, T_i \neq T_k$,
                ∀ $crit_j \in criteria(C)$;
            choose a $T_i \in Par_C$ to be $T_C$;
        else // $||Par_C|| = 1$
            the only $T \in Par_C$ becomes $T_C$;
        end if
    else // $||Par_C|| = 0$
        add VIT_C to prodList;
    end if
end for
for each declared tuple_var T do
    add VIT_T to prodList;
end for
for each declared rel_var R do
    generate set $Child_R$;
    if ∃ expression R.(criterion) in the query then
        add VIT_R to prodList;
        add VIT_R.db = VIT_$T_i$.db to selList,
            ∀ $T_i \in Child_R$;
        add VIT_R.rel_label = VIT_$T_i$.rel_label to selList,
            ∀ $T_i \in Child_R$;
    end if
end for
for each declared db_var D do
    add VIT_D to prodList;
    add VIT_D.db = VIT_$R_i$.db to selList,
    ∀ $R_i \in \mathcal{R}el_D$;
end for
for each declared variable do
    create the RRA subexpression for
    the variable's VIT
end for

// scanning of the query clauses

for each condition in the WHERE clause do
    if the condition is neither an ISA
                    nor an HASA condition then
        use the appropriate rule in Table 5;
    end if
end for
for each domain in the GROUP BY clause do
    add ADD(domain) to groupbyList;
end for
for each condition in the HAVING clause do
    use the appropriate rule in Table 5;
end for
if the SELECT clause has
  an outer FOR sub-clause then
  for each domain in the crit_list do
      add ADD(domain) to projList;
      add ADD(domain) to addrelList;
  end for
  use the (label) for the output table(s) name(s);
end if
for each select_object in the SELECT clause do
    use the appropriate rule in Table 4;
end for
for each parameter list do
    eliminate duplicate parameters in list
end for
```

Figure 19: Algorithm nD-SQLtoRRA

Expression in the nD-SQL query	List	Parameter to be added to the List
domain [AS *label*]	projList [renamList	ADD(*domain*) (ADD(*domain*), *label*)]
domain [AS *label*] FOR *crit_list*	projList addconcList addList projList [renamList	ADD(*domain*) ADD(*domain*) ADD(*domain*), $\forall$ *domain* $\in$ *crit_list* ADD(*domain*), $\forall$ *domain* $\in$ *crit_list* (ADD(*domain*), *label*)]
AGG(*domain*) [AS *label*]	projList [renamList	AGG(ADD(*domain*) (AGG(ADD(*domain*), *label*)]
AGG(*domain*) [AS *label*] FOR *crit_list*	projList addconcList addList projList [renamList	AGG(ADD(*domain*) AGG(ADD(*domain*) AGG(ADD(*domain*), $\forall$ *domain* $\in$ *crit_list* AGG(ADD(*domain*), $\forall$ *domain* $\in$ *crit_list* (AGG(ADD(*domain*), *label*)]

Table 4: Rules for select_objects in the SELECT clause

Expression in the WHERE clause	Parameter to be added to selList
$domain_1$ *Op* $domain_2$	ADD($domain_1$) *Op* ADD($domain_2$)
domain Op "value"	ADD(*domain*) *Op "value"*
"value" Op domain	*"value" Op* ADD(*domain*)

Expression in the HAVING clause	Parameter to be added to haveList
AGG($domain_1$) *Op* AGG($domain_2$)	AGG(ADD($domain_1$)) *Op* AGG(ADD($domain_2$))
AGG($domain_1$) *Op* $domain_2$	AGG(ADD($domain_1$)) *Op* ADD($domain_2$)
$domain_1$ *Op* AGG($domain_2$)	ADD($domain_1$) *Op* AGG(ADD($domain_2$))
AGG(*domain*) *Op "value"*	AGG(ADD(*domain*)) *Op "value"*
"value" Op AGG(*domain*)	*"value" Op* AGG(ADD(*domain*))

Table 5: Rules for conditions in the WHERE and HAVING clauses

PROOF:

The proof of Theorem 5.1.2 consists of 2 parts. In Part I, we show that an RRA expression following template T is equivalent to the nD-SQL semantics. Then, in Part II, we show that the nD-SQLtoRRA algorithm creates an RRA expression following template T and with the right parameters to give the correct result for Q.

Part I:

Recalling the semantics of nD-SQL we know that in general, a query result is obtained by:

(i) First instantiating variables to objects and then to logical tuples;

(ii) Then, keeping those logical tuples satisfying the conditions in the WHERE clause;

(iii) Then, aggregating tuples and keeping those satisfying the conditions in the HAVING clause;

(iv) Then, discarding extraneous dimensions;

(v) Then, separating tuples in equivalence classes based on the values in the dimensions destined to become relation criteria;

(vi) Finally, applying a merge on each equivalence class to restructure some of the logical dimensions along the row dimension.

The operations in the template can be shown to be equivalent to these steps, as follows:

(a) In the template, the task of obtaining instantiations of logical tuples is achieved by a combination of several operations. VIT subexpressions are created for each variable. For database, relation and column variables, this corresponds to instantiating the variables and obtaining their set of logical tuples. For tuple variables, the logical tuples are obtained by applying a REM_COL operation to the VIT *whenever a column variable is declared having the same db and rel range component as the tuple variable.* Then, all the tables above are joined using conditions ensuring that the logical tuples for corresponding databases, relations, columns and tuples are joined together.

In the template, the join and restructuring of VITs are split into first a Cartesian product of the VITs, then a REM_COL operation for all necessary criteria removals, then some selections. This sequence of operations is $\equiv_{nD}$-equivalent to the one dictated by the semantics, as per the Identities 5.1.1-(d) and (i);

(b) Satisfying logical tuples are obtained by applying additional selections in the third step of the template, corresponding to the conditions of the WHERE clause;

(c) The aggregated tuples are then obtained by the use of the AGG operator, followed by an additional selection to keep only those aggregated tuples satisfying the HAVING clause conditions;

(d) Discarding extraneous dimensions is then done through a projection;

(e) The last two steps, corresponding to restructurings along the relation and row dimensions for (v) and (vi) respectively, are done using the ADD_COL and ADD_REL operators, but in the opposite order to that dictated by the semantics. The result is $\equiv_{nD}$-equivalent as shown by Identity 5.1.1-(a).

Part II:

We now show that the algorithm creates the proper RRA expression for the query. We proceed according to the steps recognized in Phase I:

(a) First, we examine the process of preparing the VIT subexpressions. For database, relation and column variables, the catalog tables are queried. The RRA subexpression for database variable VIT obviously corresponds to the right instances. For VITs of relation and column variables, the use of selections based on the ISA and HASA conditions of the query, and the use of ADD_COL to create one column per criteria, ensure that valid instances are associated with the variable. As for tuple variables, the tuples from every table in the variable's range are included in the VIT. To ensure that all instantiations are also consistent, appropriate join conditions between the VITs are generated (as parameters added to the selList) in the pre-processing step.

    Once the VIT subexpressions are prepared and the appropriate selection parameters are added to the selList, the useful VITs are added as parameters to the prodList for the scalar product. The VIT of a column variable will only be added to the prodList *if there are no tuple variables declared with the same range components*. This because if there was at least one such tuple variable, a REM_COL operation will be planned for its VIT such that the column criteria will become concepts of simple columns, and conditions on the column variable's criteria will be applied to these columns. This also ensures the creation of proper logical tuples for the tuple variables;

(b) In the second step, the WHERE clause is scanned and every non-ISA, non-HASA condition is translated into a parameter in the selList for the selection, thus ensuring that only satisfying instantiations be retained;

(c) In the third step, the GROUP BY and HAVING clauses are scanned and parameters are added to the groupbyList and havList. Then, the scanning of the select_objects in the SELECT clause will add in the aggList, the last parameters necessary for aggregation;

(d) In the fourth step, the scanning of the SELECT clause ensures that for every concepts and criteria needed in the output structure, the appropriate columns are added as parameter to the projList;

(e) In the fifth and final step, the scanning of the SELECT clause will determine what restructuring needs to be executed. Parameters will be added to the addList, addconcList and addrelList, which will be used by the ADD_COL and ADD_REL operators.

Thus the algorithm does create an RRA expression which gives the correct result for the *n*D-SQL query according to the semantics described in Chapter 4. ■

Here is an example query to illustrate the translation process.

```
              SELECT    (AVG(T1.C1) FOR C1.Measure) FOR T.Date
              FROM      bse::prices -> C1, bse::prices T1
                        mse -> R, mse::R -> C2, mse::R T2
(Q14)         WHERE     C1 ISA Price AND R ISA prices AND C2 ISA Price
                        AND C1.Ticker >= 'm' AND C1.Ticker < 'n'
                        T1.Date > 10|27|97 AND T1.Date <= 11|01|97
                        AND T1.Date = T2.Date AND C1.Measure = C2.Measure
                        AND C1.Ticker = R.Ticker AND T1.C1 > T2.C2
              GROUP BY T.Date, C1.Measure
```

This query compares the values from exchanges mse and bse for the week of October 27 to November 1 1997, and for the Tickers which names start with an 'm'. The query further only takes the average of those values from exchange bse that exceed the corresponding values on the same day in exchange mse.

Following the steps of the nD-SQLtoRRA algorithm, we obtain:

After the pre-processing steps, the definitions applied to query Q14 become:
$\text{Par}_{\text{C1}} = \{\text{T1}\} \Rightarrow ||\text{Par}_{\text{C1}}|| = 1 \Rightarrow \text{T}_{\text{C1}} = \text{T1}$
$concept(\text{C1}) = \text{Price}$
$criteria(\text{C1}) = \{\text{Measure}, \text{Ticker}\}$
$\text{Par}_{\text{C2}} = \{\text{T2}\} \Rightarrow ||\text{Par}_{\text{C2}}|| = 1 \Rightarrow \text{T}_{\text{C2}} = \text{T2}$
$concept(\text{C2}) = \text{Price}$
$criteria(\text{C2}) = \{\text{Measure}\}$
$\text{Child}_{\text{R}} = \{\text{T2}\}$

Some parameter lists are initialised as such:
remList = {VIT_T1.Measure, VIT_T1.Ticker, VIT_T2.Measure}
remconcList = {VIT_T1.Price, VIT_T2.Price}
prodList = {VIT_T1, VIT_T2, VIT_R}
selList = {VIT_T2.db = VIT_R.db, VIT_T2.rel_label = VIT_R.rel_label}

The RRA subexpressions for VITs are given by the following expressions:

$$\text{VIT\_R} = \rho_{\text{value}\to\text{criteria}}\{\text{ADD\_COL}_{\text{criteria}\to\text{value}}(\Pi_{db,rel\_label,criteria,value}[\sigma_{db='mse' \,\wedge\, rel\_concept='prices'}\{\text{dbscheme} \bowtie_{relid=relid} \text{relschemes} \bowtie_{relid=id} \text{criteria}\}])\}$$

$$\text{VIT\_C1} = \rho_{\text{value}\to\text{criteria}}\{\text{ADD\_COL}_{\text{criteria}\to\text{value}}(\Pi_{db,rel\_label,attr\_label,criteria,value}[\sigma_{db='bse' \,\wedge\, rel_label='prices' \,\wedge\, attr\_concept='Price'}\{\text{dbscheme} \bowtie_{relid=relid} \text{relschemes} \bowtie_{attrid=id} \text{criteria}\}])\}.$$

$$\text{VIT\_C2} = \text{ADD\_COL}_{\text{criteria}\to\text{value}}(\Pi_{db,rel\_label,attr\_label,criteria,value}[\sigma_{db='mse' \,\wedge\, attr\_concept='Price'}\{\text{dbscheme} \bowtie_{relid=relid} \text{relschemes} \bowtie_{attrid=id} \text{criteria}\}]).$$

$$\text{VIT\_T1} = \bigcup_{rel_i=\text{bse::prices}} (\{rel_i\}) = \text{bse :: prices}$$

$$\text{VIT\_T2} = \bigcup_{rel_i=\text{mse::ibm, mse::ms, ...}} (\{rel_i\})$$

After scanning the query clauses, some values have been added to the various parameter lists as follows:

WHERE: add to selList {VIT_T1.Ticker >= 'm', VIT_T1.Ticker < 'n'
VIT_T1.Date > 10|27|97, VIT_T1.Date < 11|01|97,
VIT_T1.Date = VIT_2.Date, VIT_T1.Measure = VIT_T2.Measure,
VIT_T1.Ticker = VIT_R.Ticker, VIT_T1.Price > VIT_T2.Price}

GROUP BY: groupbyList = {VIT_T1.Date, VIT_T1.Measure}

SELECT: projList = {VIT_T1.Measure, VIT_T1.Date, AVG(VIT_T1.Price)}

addrelList = {VIT_T1.Date}
concList = {AVG(VIT_T1.Price)}
addList = {VIT_T1.Measure}
aggList = {AVG(VIT_T1.Price)}

The list selList will thus end up as:
selList = {VIT_T2.db = VIT_R.db, VIT_T2.rel_label = VIT_R.rel_label,
VIT_T1.Ticker >=' m', VIT_T1.Ticker <' n',
VIT_T1.Date > 10|27|97, VIT_T1.Date < 11|01|97,
VIT_T1.Date = VIT_T2.Date, VIT_T1.Measure = VIT_T2.Measure,
VIT_T1.Ticker = VIT_R.Ticker, VIT_T1.Price > VIT_T2.Price}

The final RRA expression is obtained simply by using template T with the above parameters, plus the RRA subexpressions for the VITs whose names appear in parameter list prodList.

It is very important to note that *all* VIT*s have to be instantiated, not just those in the prodList.* Those not in the prodList will not be used in the Cartesian product. On the other hand, they play an important role in optimisation.

5.1.3 Optimisation

Many opportunities for optimisation arise from the use of RRA in processing *n*D-SQL queries. Remember that tuple variables must be instantiated by querying various local sources. By taking advantage of the downward compatibility of RRA to RA and by using the various identities presented in Theorem 5.1.1, we find that there exists the opportunity to push to remote sources some computations. Is can be possible, for example, to push some selections, projections, aggregations and even joins. Eventually, *although our architecture does* **not** *demand it.* if a local database supports restructuring, it would be possible to also push some of those operations to the local system.

A preliminary step in optimising the computations consists in ordering the instantiation of variables and using the technique of *sideways information passing* (sip) first proposed in [Bee87]. This becomes particularly important in order to determine what database and/or relation to access to instantiate some tuple variable. Equally important is the possibility of passing bindings from a first instantiated variable to the query instantiating a second one. This opportunity arises when a join is called for between tables originating from two distinct sources. In some situations, the instantiation of the second variable should be delayed until the values of the join column(s) obtained from the first variable's instantiations can be passed as bindings. These bindings would be passed on as selections in a SQL query.

In addition to using the identities of Theorem 5.1.1, an RRA optimiser can also use the classical identities involving the operators of RA. Also, the following equivalences arising from the symmetry between the restructuring operators REM_COL and ADD_COL, can be used for optimisation purposes:

Theorem 5.1.3 *RRA expression equivalences.*

- $\text{ADD\_COL}_{p_1 \rightarrow p_2}$ [$\text{REM\_COL}_{p_3 \leftarrow p_2}$ $(Table)$] $\equiv_{nD}$ $\text{ADD\_COL}_{p_1 - p_3 \rightarrow p_2}$ $(Table)$, *if* $p_3 \subset p_1$;

- $\text{ADD\_COL}_{p_1 \rightarrow p_2}$ [$\text{REM\_COL}_{p_3 \leftarrow p_2}$ $(Table)$] $\equiv_{nD}$ $\text{REM\_COL}_{p_3 - p_1 \leftarrow p_2}$ $(Table)$, *if* $p_1 \subset p_3$;

- $\text{ADD\_COL}_{p_1 \rightarrow p_2}$ [$\text{REM\_COL}_{p_3 \leftarrow p_2}$ $(Table)$] $\equiv_{nD}$ *Table, if* $p_1 = p_3$;

PROOF: Obvious

Another set of optimisation rules rely on the following heuristic:

Heuristic 5.1.1 *It is in general more efficient to perform join or restructuring on fewer tuples, albeit they be wider.*

Since *in general* ADD_COL lowers the number of tuples and REM_COL increases it, the following additional heuristics can be derived:

Derived Heuristics 5.1.1

a) $\text{REM\_COL}_{p_3 \leftarrow p_4}$ [$\text{ADD\_COL}_{p_1 \rightarrow p_2}$ $(Table)$] *is more efficient than* $\text{ADD\_COL}_{p_1 \rightarrow p_2}$ [$\text{REM\_COL}_{p_3 \leftarrow p_4}$ $(Table)$].

b) If $\bowtie_{p_1}$ *and* $\text{REM\_COL}_{p_2 \leftarrow p_3}$ *can commute and* p_2 *only refers to* $Table_2$, *then* $\text{REM\_COL}_{p_2 \leftarrow p_3}$ [$Table_1 \bowtie_{p_1} Table_2$] *is more efficient than* $Table_1 \bowtie_{p_1}$ [$\text{REM\_COL}_{p_2 \leftarrow p_3}$ $(Table_2)$], *provided the join selectivity is high.*[1]

[1] Recall, the higher the join selectivity, the fewer the tuples that result from the join.

c) If $\bowtie_{p_1}$ *and* $\text{ADD\_COL}_{p_2 \rightarrow p_3}$ *can commute and* p_2 *and* p_3 *refer only to* $Table_2$*, then* $Table_1 \bowtie_{p_1}$ [$\text{ADD\_COL}_{p_2 \rightarrow p_3}$ $(Table_2)$] *is more efficient than* $\text{ADD\_COL}_{p_2 \rightarrow p_3}$ [$Table_1 \bowtie_{p_1} Table_2$], *provided the join selectivity is low.*

d) $\text{REM\_COL}_{p_3 \leftarrow p_4}$ [$\text{AGG}_{p_1, p_2 - p_3}$ $(Table)$] *is more efficient than* AGG_{p_1, p_2} [$\text{REM\_COL}_{p_3 \leftarrow p_4}$ $(Table)$]

Another form of optimisation would be to take advantage of what we call "interleaving". Interleaving is the efficient implementation of a series of operators that are often called for in sequence, similar to the way join is a more efficient implementation of Cartesian product 'interleaved' with selection. In RRA, two such series of operations can be pinpointed:

1. A selection applied to the values of a criterion of a complex column, without any restructuring of the criterion being otherwise necessary, can be implemented more efficiently than by first removing the criterion, selecting on it, and adding it back.

2. A selection applied to the values of the concept of a complex column, without any restructuring of the columns being otherwise necessary, can be implemented more efficiently than by removing all criteria of that complex column, selecting on the concept and adding all the criteria back.

Let us define two new operators, Π^* and σ^* that capture the sequence of operations 1 and 2 respectively.

Definition 5.1.7 (Π^*) *The operation* $\Pi^*_{projList,\ selList}(\texttt{rel})$*, projList being a list of column concepts and column labels of* `rel`*, and selList being a list of conditions involving criteria of complex columns of* `rel`*, applied to a relation with name* `rel`*, has the following effect. Let r be any instance of the relation name* `rel` *in the database. Then, corresponding to each such relation r, the operation produces an output relation r′ satisfying the following conditions.*

- *The column labels of* r' *are* $cols(r') = \{A \mid A \in cols(r) \wedge (concept(A) \in projList \vee A^2 \in projList\} \cup \{C \mid C$ *is a complex column of* $r \wedge C's$ *criteria values fulfill the conditions in* $selList\}$

- *The instance of* r' *consists of a set of tuples over* $cols(r')$*, defined as* $inst(r') = \{t \mid \exists s \in r \wedge t[c_i] = s[c_i], \forall c_i \in cols(r')\}$

[2] as a label

Definition 5.1.8 (σ^*) *The operation* $\sigma^*_{selList}$(rel), *selList being a list of conditions involving concepts of complex columns of* rel, *applied to a relation with name* rel, *has the following effect. Let r be any instance of the relation name* rel *in the database. Then, corresponding to each such relation r, the operation produces an output relation r', with the same schema as r, satisfying the following conditions.*

- *The columns of r' are $cols(r') = cols(r)$. Let C_s be the set of simple columns of r' and let C_c be the set of complex columns of r'.*

- *The instance of r' consists of a set of tuples over $cols(r')$, defined as $inst(r') = \{t \mid \exists s \in r : t[c_i] = s[c_i], \forall c_i \in C_s \wedge t[c_j] = s[c_j], \forall c_j \in C_c$ s.t. $s[c_j]$ fulfils the conditions in the selList $\wedge\ t[c_k] = null, \forall c_k \in C_c$ s.t. $s[c_k]$ does not fulfill the conditions in the selList $\wedge\ \exists c_j \in C_c$ s.t. $s[c_j]$ fulfils the conditions in the selList$\}$*

As an example of the use of the interleaving operators, the result of the operation $\Pi^*_{\texttt{Date,Ticker>'m'}}$(bse :: prices) is given in Figure 20 while the result of operation $\sigma^*_{\texttt{Price>50.00}}$(bse :: prices) is given in Figure 21.

Date	open_ms	open_...	...	close_ms	close_...
10|27|97	50.23	...	...	48.54	...
...	...	...	...	...	...
11|01|97	44.60	...	...	46.17	...

Figure 20: Result of operation $\Pi^*_{\texttt{Date,Ticker>'m'}}$(bse :: prices)

Date	open_ibm	open_ms	open_...	...	close_ibm	close_ms	close_...
10|27|97	63.67	50.23	...	...	62.56	null	...
...	...	...	...	...	...	...	...
11|01|97	65.23	null	...	...	63.05	null	...

Figure 21: Result of operation $\sigma^*_{\texttt{Price>50.00}}$(bse :: prices)

Figure 22 gives an algorithm for implementing operator Π^* while Figure 23 gives an algorithm for implementing operator σ^*.

The intuition behind the algorithm implementing operator Π^* is as follows: we project the tuples of the relation, keeping those columns that have their concepts in selList *or* that are instance of complex columns such that their criteria values fulfill the conditions in the selList.

The intuition behind the algorithm implementing operator σ^* consists in scanning each tuple of an input table. If an instance of a complex column has its concept in

```
INPUT: Table $\mathcal{T}_i$ on which to project, with schema $\mathcal{R}_i$
         Set projList of concepts of columns of $\mathcal{R}_i$ to project
         Set selList of conditions on criteria of complex columns of $\mathcal{R}_i$

OUTPUT: Table $\mathcal{T}_o$ of selected tuples, with schema $\mathcal{R}_o$


$\mathcal{R}_o = \{\}$;
for each column $r_i \in \mathcal{R}_i$ do
    if $concept(r_i) \in$ projList then
      $\mathcal{R}_o = \mathcal{R}_o \cup \{r_i\}$;
    else if $r_i$ is an instance of a complex column of $\mathcal{T}_i$ then
           if $r_i$'s criteria values fulfill the conditions in selList then
             $\mathcal{R}_o = \mathcal{R}_o \cup \{r_i\}$;
           end if
    end if
end for

for each tuple $t_i \in \mathcal{T}_i$ do
    add $t_i[\mathcal{R}_o]$ to $\mathcal{T}_o$
end for
```

Figure 22: Algorithm for operator Π^*

```
INPUT: Table $\mathcal{T}_i$ on which to select, with schema $\mathcal{R}_i$
         Set selList of conditions on concepts of complex columns of $\mathcal{R}_i$

OUTPUT: Table $\mathcal{T}_o$ of projected tuples, with schema $\mathcal{R}_i$


for each tuple $t_i \in \mathcal{T}_i$ do
    let $t_o$ be a new tuple for output table $\mathcal{T}_o$
    good_tuple = false;
    for each $r_i \in \mathcal{R}_i$ s.t. $r_i$ is an instance of a complex column do
        if $t_i[r_i]$ fulfills the conditions in the selList then
            $t_o[r_i] = t_i[r_i]$;
            good_tuple = true;
        else
            $t_o[r_i]$ = NULL;
        end if
    end for

    if good_tuple then
      for each $r_i \in \mathcal{R}_i$ s.t. $r_i$ is a simple column do
            $t_o[r_i] = t_i[r_i]$;
      end for
      add $t_o$ to $\mathcal{T}_o$;
    end if
end for
```

Figure 23: Algorithm for operator σ^*

the selList, if the value in that column does not fulfill the conditions in the selList, it is replaced by a null value. If all complex column values are replaced by nulls, the tuple is dropped, otherwise it is kept.

Since in general the local sources will not support restructuring operations, but will support classical RA operations, those may be pushed to the local sites. Thus, it will often be the case that the first operation to be executed on the data transferred from a local source will be a restructuring. Given this, another interesting interleaving to be implemented would be the restructuring of data as it is received from a local source, rather than receiving all the data, storing it as a table and then restructuring it. This would be particularly interesting in cases where great volumes of data are transferred and have to be restructured.

Another opportunity for optimisation in the nD-SQL setting concerns redundant data transferred from the same remote course. Some queries may involve multiple tuple variables for which the ranges overlap. An optimisation strategy should be employed in order to eliminate (or at least minimise) the quantity of redundant data transferred. One idea would be to define one or more "envelopes" such that in a given situation (parametrised by the selections to apply, the projections to make and any available statistical information about the remote source), the envelope(s) would represent the optimal SQL query or queries to be sent to the remote database in order to transfer *all* the information needed without (or without much) redundancy.

The data corresponding to each tuple variable may differ both in the fields selected (i.e. the projections that can be pushed to the remote source) and the conditions on the various dimensions. Also, the specific cost model has to be taken into account: should we optimise the amount of data transfered or the CPU time needed to compute the envelope *versus* the one needed to compute each individual data set. Probably a combination of both. Thus the problem of defining a proper envelope is a non trivial one. This problem has also arisen in other settings involving the querying of data from different sources ([SV98]).

5.2 Processing of Queries With Dimension Variables

The most interesting (and challenging) class of queries of this kind are the ones which involve aggregation. The key idea in their processing is recognising that they involve computing a subset of group-bys from the cube lattice. Such computations are referred to as *partial cubes* [Agar+96, ZDN97]. ROLLUP is a common example of a partial cube. See Example 3.2.4 for another interesting example of a partial cube. The references [Agar$^+$96, ZDN97] discuss how algorithms for computing the CUBE can be adapted for computing partial cubes. Optimisation of partial cubes is a topic of its own interest and is orthogonal to this thesis. It should mainly be observed that queries with dimension variables and aggregation may in general involve: (i) computing a partial cube, and (ii) computing multiple visualisations of the result. The processing of such queries can be organised as follows.

1. Identify the precise partial cube to be computed, by instantiating the dimension variables in the query;
2. Apply any fast algorithm in the literature for computing the partial cube. These algorithms can be made more efficient by taking advantage of the implicit grouping provided by column and relation criteria;
3. Apply the required restructuring operations for each group-by computed in step 2.

An interesting research problem left for future work is: how to interleave the computation of the partial cube with the required restructuring for each group-by in the partial cube.

Chapter 6

Implementation

In this chapter, we discuss the performance results obtained by testing various heuristics developed for RRA expression optimisation. Those performance evaluations are crucial to understand the tradeoffs between alternate strategies for processing nD-SQL queries and developing an nD-SQL Server, currently an ongoing activity. To set the context, Section 6.1 discusses our work on the nD-SQL Server, while Section 6.2 presents and discusses the performance results.

6.1 Implementation Details

As Figure 13 indicates, the implementation of an nD-SQL Server can be realized as an external module, independent of the databases in the federation. The Server's main components are: a Query Interface, a resident database engine storing the catalog database, a Translator to go from nD-SQL to RRA, an RRA Expression Optimiser and an RRA Expression Executor.

Some explanations about the system's architecture are in order. Once a query is accepted by the Interface, the Translator creates an $\equiv_{nD}$-equivalent RRA expression which is sent to the Optimiser for a first pass. Then, The Translator builds the SQL queries that will create the tuple_var VITs and those queries are submitted to local databases, using sip to determine the order of submission, and passing parameters from one result to another query. After all the VITs are instantiated, the Optimiser does a second pass optimisation of the RRA expression. The Executor then executes the optimised expression, using restructuring operations, and presents the final result to the user.

Our implementation work has focused on the Expression Executor and the Optimiser. We have done some preliminary work on the Translator and the Query Interface modules. These are not discussed in detail in this Thesis. The work that remains to be done is discussed in Chapter 8.

The current implementation was done on the PC platform, under Windows 95 with the RRA operators coded in C++ and using the MS Access database engine.

6.2 Performance Results

In order to verify the validity of some of the heuristics we derived in Section 5.1.3, we carried out some experiments using our implementation of the RRA operators. The performance results for three heuristics are presented in the next section. However, before presenting those results, some preliminary statements are necessary.

6.2.1 Preliminary Statements

First, we need to identify the set of parameters that can influence the efficiency of the operators involved in the performed tests.

An important parameter is the number of tuples in the input tables for each operator. Thus, the number of tuples in the base input tables is our first parameter. Then, we need to parametrise the number of tuples in the output of each operator.

For the Join operator, the *join selectivity* is the measure of the number of tuples in the output. We use the following definition of join selectivity:

$$\text{join selectivity} = \frac{\text{nbr of tuples in } Table_1 \times \text{nbr of tuples in } Table_2}{\text{nbr of tuples in } Table_1 \bowtie Table_2}$$

In all our tests, sequences of operations involving Joins also involve some restructuring operation. Of the two input tables to each sequence, when varying the input size, we thus vary only the size of that table involved in the restructuring. In that setting, the impact of the join selectivity can be better understood by using a normalised selectivity. We define the *average join selectivity with respect to a table $Table_i$* as:

$$\text{avg join selectivity w.r.t. } Table_i = \frac{\text{nbr of tuples in } Table_i}{\text{nbr of tuples in } Table_1 \bowtie Table_2}$$

This normalisation is defined such that the avg join selectivity w.r.t. $Table_i$ is 1 when on average, there is one tuple in the output of the Join per tuple in the input table $Table_i$. Whenever the result of the Join contains more tuple than table $Table_i$, the normalised selectivity is < 1, while it is > 1 when the result of the Join contains less tuple than table $Table_i$.

Another important property of the data is the *average compactness* of a table with respect to a restructuring. This is a measure of the average number of logical tuples that are merged together in a certain structural representation of the data compared to another representation.

Formally, the *average compactness* of a table with respect to a criteria set $\mathcal{C}$ and a set of concepts $\mathcal{A}$ is

$$\text{avg compactness} = \frac{\text{nbr of tuples in the least compact representation w.r.t. } \mathcal{C} \text{ and } \mathcal{A}}{\text{nbr of tuples in the most compact representation w.r.t. } \mathcal{C} \text{ and } \mathcal{A}}$$

where the most compact representation is the one for which all $c_i \in \mathcal{C}$ are criteria of the concepts in $\mathcal{A}$, and the least compact representation is the one for which all $c_i \in \mathcal{C}$ are concepts of simple columns.

As an example of this, refer to Figure 14 which shows the result of performing operation $\text{ADD\_COL}_{\text{Measure}\rightarrow\text{Price}}(\texttt{nyse} :: \texttt{prices})$. Say there are 5 distinct `Measure` values in `nyse::prices` and `ny2t::prices`, and moreover suppose that there are `Price` values for all these `Measures` for each `Ticker` and each `Date`. Then, the compactness of both `nyse` and `ny2t` with respect to the criterion `Measure` and the concept `Price` is 5.

Several other parameters affect the efficiency of our restructuring operators. Among them: the number of criteria involved, the number of concepts of complex columns involved, the compactness *variance* (in conjunction with the average compactness), and the size of the input, intermediary, and output data sets compared to the amount of available main memory. In a distributed environment, where an nD-SQL server is used to query simultaneously several distinct data sources over a network, the network traffic, the size of the various pipes and the cost models associated with each source (which can all be different!) must be factored in by the optimiser. Testing the efficiency of query evaluation strategies with respect to each of these parameters, and combination thereof, is part of our ongoing work, to be discussed in Chapter 8.

6.2.2 Testing Methodology

Tests were conducted in pairs on a PC. Both possible orderings of operations for a heuristic were tested in the same test run, the ordering expected to run faster (according to the heuristic) running second. In this way, if the slowest and most demanding ordering of operations pushed the memory usage over the RAM threshold, the faster ordering should also be slowed by the use of swapping.

The data sets used were manually generated for each pair of tests in order for them to have the necessary characteristics (avg compactness, avg selectivity, etc.)

Each pair of tests was run several times, the results averaged over the number of tests. The variance of the results in each case was not significant and is not reported in the following section.

6.2.3 Results

Of the four heuristics derived in Section 5.1.3, performance evaluations of the first three have been performed. Note that for all the results presented in this section, the times given are in seconds.

The results are as follows:

ADD_COL vs REM_COL

Our first set of experiments tested heuristic 5.1.1 a). The varying parameter was the number of tuples in the input table. The compactness of the input table was fixed at a value of 12.

The results of the tests are summarised in Table 6. A graphical representation of the results is presented in Figure 24.

Nbr tuples in Table	Time for REM_COL (ADD_COL (Table))	Time for ADD_COL (REM_COL (Table))	Speed-up
10	18.25	17.25	0.95
100	48.50	148.00	3.05
300	331.67	721.33	2.17
1000	2999.00	7713.00	2.57
3000	26079.00	72933.50	2.80
5000	68372.00	197621.00	2.90

Table 6: Performance of ADD_COL vs REM_COL for varying number of input tuples

We can see that, as expected, the execution times of both sequence of operations is proportional to the number of tuples in the input. Also, as the number of input tuples increases, we observe a general increase in the speed-up of the first sequence

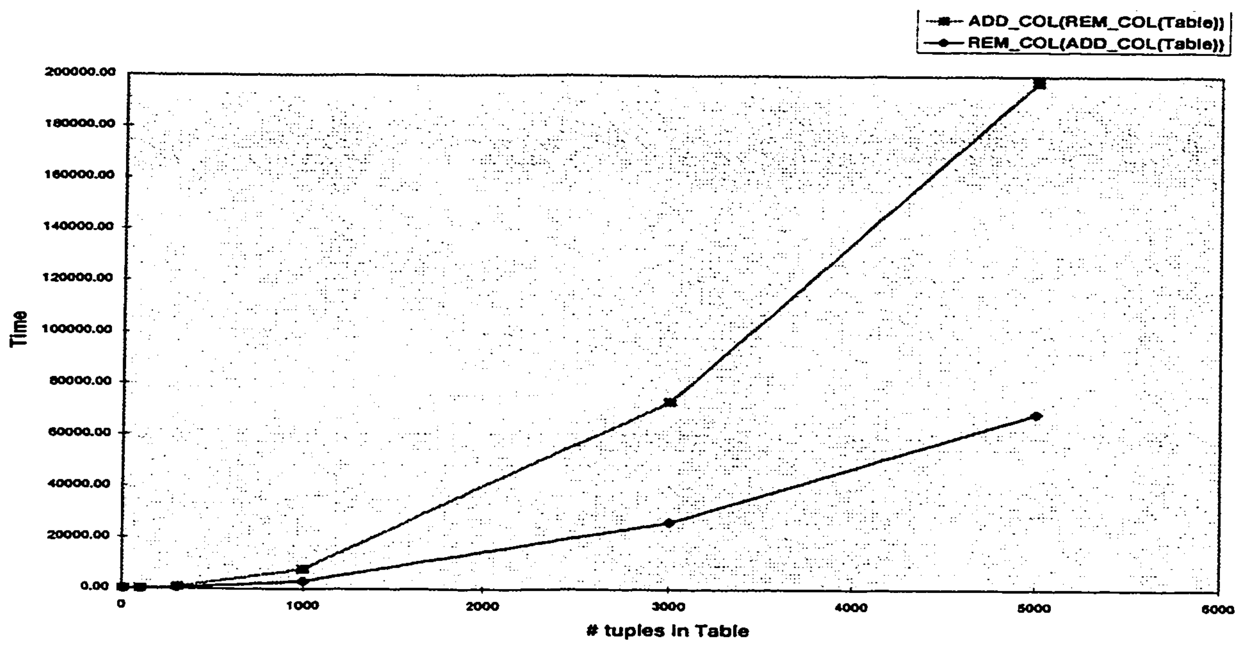

Figure 24: Performance of ADD_COL vs REM_COL for varying number of input tuples

of operations with respect to the second. This is in accordance with our heuristic. Why is REM_COL(ADD_COL(Table)) a more efficient sequence of operations? Simply because the input to both operators are smaller than in the reverse sequence. By applying ADD_COL first to the input table, the number of tuples then given as input to the REM_COL operator is reduced, albeit each of these tuples is wider. The tuples contain the same information, but some redundancy is eliminated by putting data in schema positions since the data value will be stored only once (in the schema) instead of appearing appearing in several tuples. A similar observation applies to the input of the ADD_COL operator, which is smaller when this operation is applied to the input table then when its input is the result of the REM_COL operator.

REM_COL vs Join

The second set of experiments tested heuristic 5.1.1 b). Here, 3 distinct parameters were varied. In one series of tests, we varied the join selectivity while the avg compactness and the number of input tuples were kept constant. In another series of tests, we varied instead the avg compactness while the other two parameters were kept constant. Finally, for the last series of tests comparing Join and REM_COL, we varied the number of tuples of input table $Table_2$ while keeping the other parameters constant.

Varying the join selectivity

For the series of tests in which we varied the join selectivity, the other parameters were fixed at the following values: avg compactness of $Table_2$: 4; nbr tuples in $Table_1$: 18; nbr tuples in $Table_2$: 2160.

The results of the tests are summarised in Table 7. A graphical representation of the results is presented in Figure 25.

Avg join selectivity w.r.t. $Table_2$	Time for REM_COL ($Table_1 \bowtie Table_2$)	Time for $Table_1 \bowtie$ (REM_COL ($Table_2$))	Speed-up
1/4	1039.00	419.00	0.40
1/1	305.66	352.00	1.15
12/1	30.66	388.00	12.65
36/1	15.33	384.00	25.05
142/1	8.66	373.00	43.07

Table 7: Performance of Join vs REM_COL for varying avg join selectivity

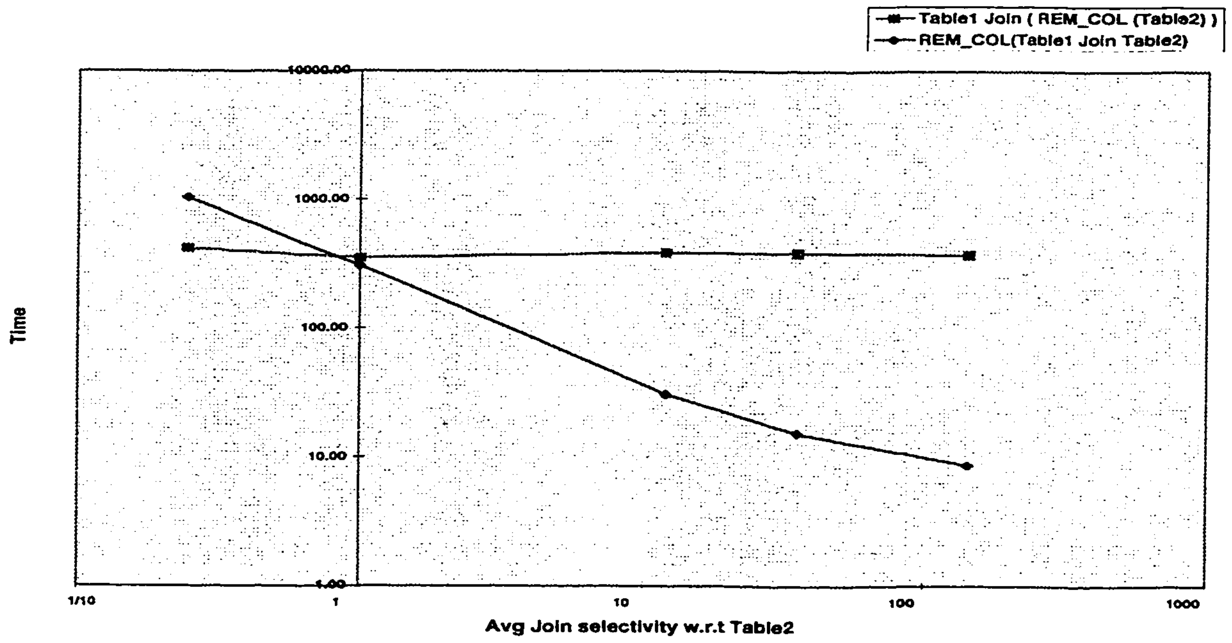

Figure 25: Performance of Join vs REM_COL for varying avg join selectivity

As predicted by the heuristic, past a threshold of high selectivity, the sequence Join followed by REM_COL is more efficient than the the the reverse sequence. This follows from the fact that when the selectivity is high enough, the number of tuples produced by the Join is smaller than the number of tuples in $Table_2$, thus the REM_COL operation receives fewer tuples as input. As the avg join selectivity increases, so does the speed-up since the input to the REM_COL operation gets smaller. We would expect the high selectivity threshold to be reached when the avg selectivity w.r.t. $Table_2 > 1$. On the other hand, the result of the REM_COL operation applied to $Table_2$ is larger than $Table_2$ itself, which means a smaller input to the Join in the sequence REM_COL followed by Join than in the reverse sequence. The threshold is therefore reached when the avg selectivity w.r.t. $Table_2$ is slightly lower than 1.

There were no indices defined on the input tables used for our experiments. This explains the flatness of the curve for the sequence REM_COL followed by Join: the time needed to execute the Join was independent of the selectivity. However, in general, Joins can make use of indices to be much more efficient, which would benefit both sequence of operations.

Varying the number of input tuples

For the series of tests in which we varied the number of tuples in $Table_2$, the other parameters were fixed at the following values: avg compactness: 4; avg join selectivity w.r.t. $Table_2$: 12; nbr tuples in $Table_1$: 18.

The results of the tests are summarised in Table 8. A graphical representation of the results is presented in Figure 26.

Nbr tuples in $Table_2$	Time for REM_COL ($Table_1 \bowtie Table_2$)	Time for $Table_1 \bowtie$ (REM_COL ($Table_2$))	Speed-up
108	6.66	28.33	4.25
540	12.33	81.33	6.60
1080	19.00	153.00	8.05
2160	30.66	388.00	12.65
4320	107.00	805.00	7.52
6480	210.00	1467.00	6.99
10800	330.00	3426.00	10.38

Table 8: Performance of Join vs REM_COL for varying nbr of input tuples

As predicted by the heuristic, the sequence Join followed by REM_COL is more efficient than the reverse sequence. We would, however, have expected the speed-up of the sequence Join followed by REM_COL with respect to the reverse sequence to increase with the number of tuples. The observed behaviour does not correspond to our expectations and further experiments will have to be performed.

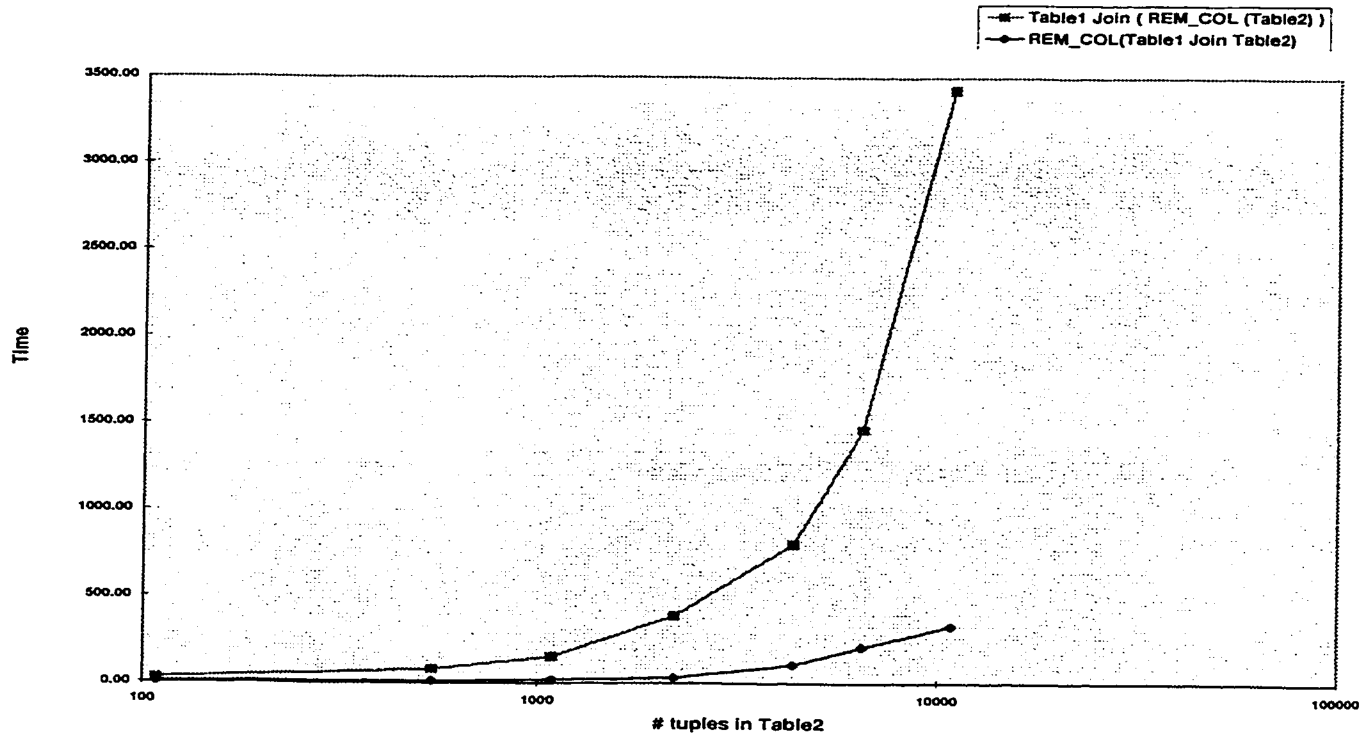

Figure 26: Performance of Join vs REM_COL for varying nbr of input tuples

Varying the avg compactness

For the series of tests where we vary the avg compactness, the other parameters were fixed at the following values: avg join selectivity w.r.t. to $Table_2$: 12; nbr tuples in $Table_1$: 18; nbr tuples in $Table_2$: 2160.

The results of the tests are summarised in Table 9. A graphical representation of the results is presented in Figure 27.

Avg compactness of REM_COL	Time for REM_COL ($Table_1 \bowtie Table_2$)	Time for $Table_1 \bowtie$ (REM_COL ($Table_2$))	Speed-up
1	7.67	49.33	6.43
4	27.33	217.00	7.94
6	47.67	359.67	7.54
10	137.00	1088.50	7.95
20	269.00	2075.00	7.71
40	1278.00	13244.50	10.36

Table 9: Performance of Join vs REM_COL for varying avg compactness

These results are particularly interesting since they fall outside the bounds of our heuristics. Figure 27 shows how the compactness affects the efficiency of the

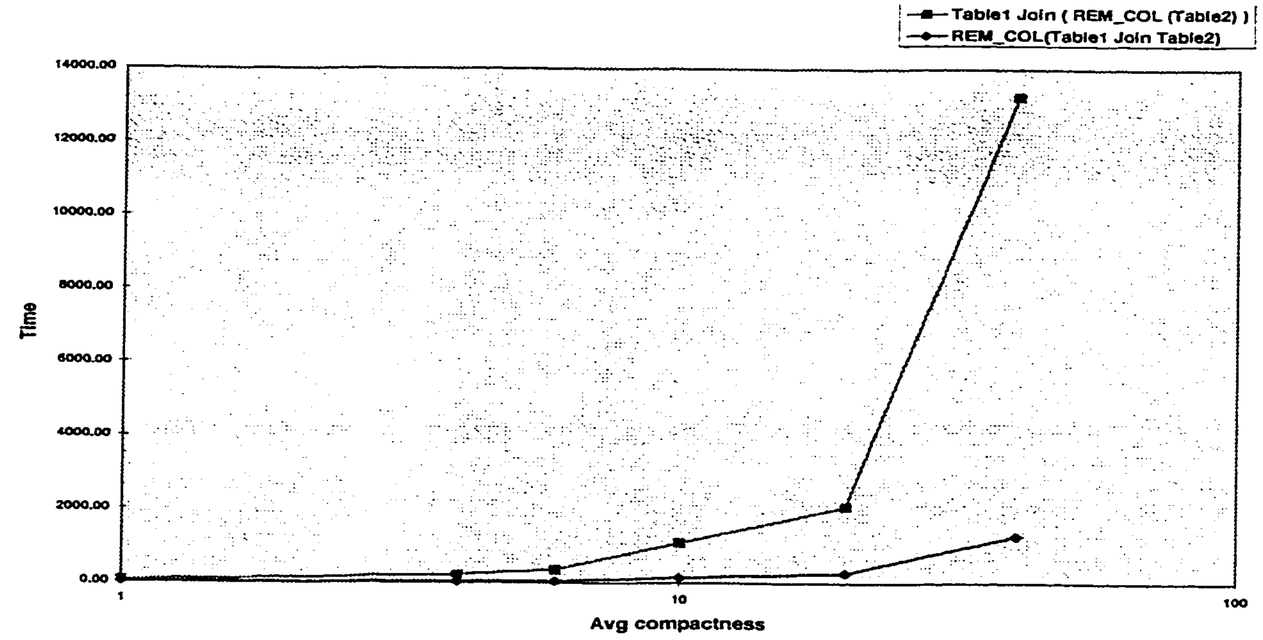

Figure 27: Performance of Join vs REM_COL for varying avg compactness

REM_COL operation. We observe that the speed-up increases in general with the compactness. This behaviour corresponds to the following intuition: As the average compactness increases, the REM_COL operation breaks down each input tuple into more output tuples, which not only is a costlier operation but also passes a larger input to the Join in the sequence REM_COL followed by Join.

ADD_COL vs Join

Our last set of experiments tested heuristic 5.1.1 c). We varied the same three parameters as before, with one parameter varied for each series of tests. The parameters were: the join selectivity, the avg compactness and the number of input tuples.

Varying the join selectivity

For the series of tests where we varied the join selectivity, the other parameters were fixed at the following values: avg compactness of ADD_COL: 4; nbr tuples in $Table_1$: 18; nbr tuples in $Table_2$: 432.

The results of the tests are summarised in Table 10. A graphical representation of the results is presented in Figure 28.

As predicted by the heuristic, past a low selectivity threshold, the sequence Join followed by ADD_COL becomes much less efficient than the reverse sequence. This follows from the fact that when the selectivity is low enough, the Join produces more tuples than the number of tuples in $Table_2$, thus passing on a larger input to the ADD_COL operation. As the avg join selectivity decreases, the speed-up increases since the input to the ADD_COL operation gets larger. We would expect the low selectivity threshold to be reached when the avg selectivity w.r.t. $Table_2 < 1$. On the other hand, the result of the ADD_COL operation applied to $Table_2$ is smaller than $Table_2$ itself, which means a smaller input to the Join in the sequence ADD_COL followed by Join than in the reverse sequence. The threshold is therefore reached when the avg selectivity w.r.t. $Table_2$ is slightly higher than 1.

Avg join selectivity w.r.t. $Table_2$	Time for $Table_1 \bowtie$ (ADD_COL ($Table_2$))	Time for ADD_COL ($Table_1 \bowtie Table_2$)	Speed-up
3/1	27.33	11.66	0.42
1/1	27.00	34.33	1.27
1/6	28.66	407.00	14.20
1/10	27.00	1171.00	43.97
1/18	30.00	3246.50	108.22
1/40	27.00	17279.00	639.96
1/80	29.50	75806.50	2569.71

Table 10: Performance of Join vs ADD_COL for varying avg join selectivity

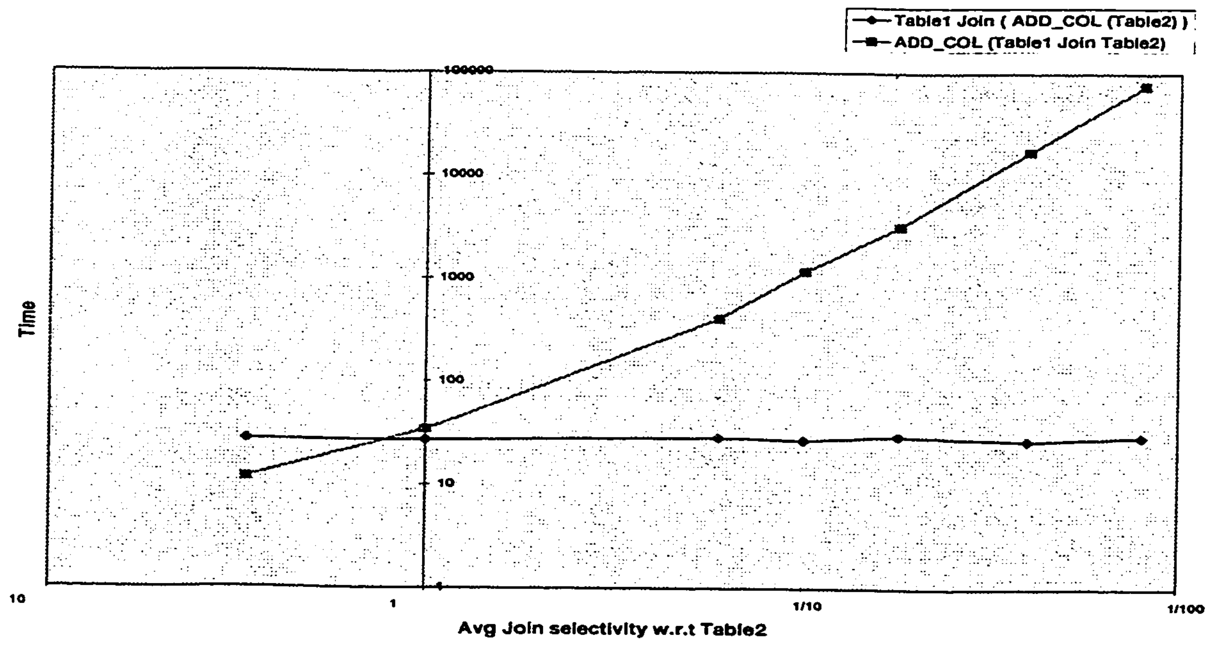

Figure 28: Performance of Join vs ADD_COL for varying avg join selectivity

Again, the flatness of the curve for the sequence ADD_COL followed by Join is explained by the absence of indices defined on the input tables.

Varying the number of input tuples

For the series of tests where we varied the number of tuples in $Table_2$, the other parameters were fixed at the following values: avg compactness of ADD_COL: 1/4; avg join selectivity w.r.t. $Table_2$: 1/6; nbr tuples in $Table_1$: 18.

The results of the tests are summarised in Table 11. A graphical representation of the results is presented in Figure 29.

Nbr tuples in $Table_2$	Time for $Table_1 \bowtie$ (ADD_COL ($Table_2$))	Time for ADD_COL ($Table_1 \bowtie Table_2$)	Speed-up
48	8.66	21.66	2.50
432	48.66	407.00	8.36
1296	103.00	3224.66	31.31
2592	282.00	13646.00	48.39
4320	731.00	37906.66	51.86
5184	939.50	57183.50	60.87

Table 11: Performance of Join vs ADD_COL for varying nbr of input tuples

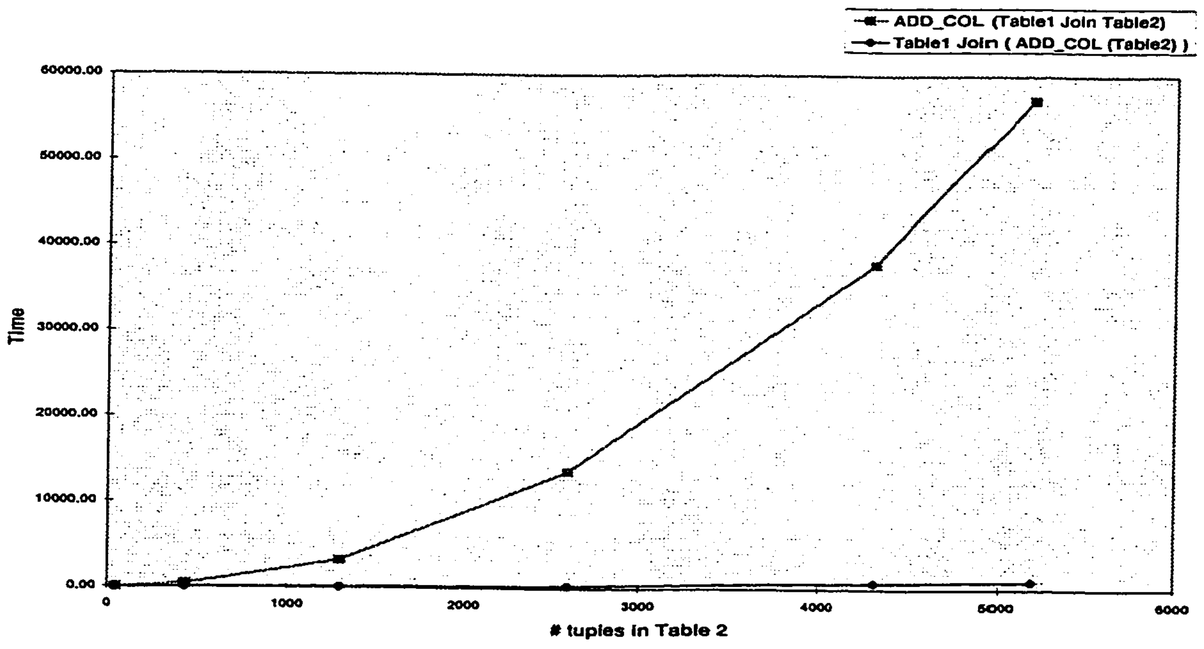

Figure 29: Performance of Join vs ADD_COL for varying nbr of input tuples

As predicted by the heuristic, the sequence ADD_COL followed by Join is more efficient than the reverse sequence. Moreover, the speed-up of the sequence ADD_COL followed by Join with respect to the reverse sequence increases with the number of input tuples. The intuition behind this behaviour is that for a fixed Join selectivity, the output of a Join increases with the size of the input.

Varying the avg compactness

For the series of tests where we varied the avg compactness, the other parameters were fixed at the following values: avg join selectivity w.r.t. to $Table_2$: 1/6; nbr tuples in $Table_1$: 18; nbr tuples in $Table_2$: 1000.

The results of the tests are summarised in Table 12. A graphical representation of the results is presented in Figure 30.

Avg compactness of ADD_COL	Time for $Table_1 \bowtie$ (ADD_COL ($Table_2$))	Time for ADD_COL ($Table_1 \bowtie Table_2$)	Speed-up
1	55.66	1292.66	23.22
4	30.66	407.00	13.27
9	31.00	295.66	9.53
48	76.00	350.33	4.60
72	158.00	753.33	4.76

Table 12: Performance of Join vs ADD_COL for varying avg compactness

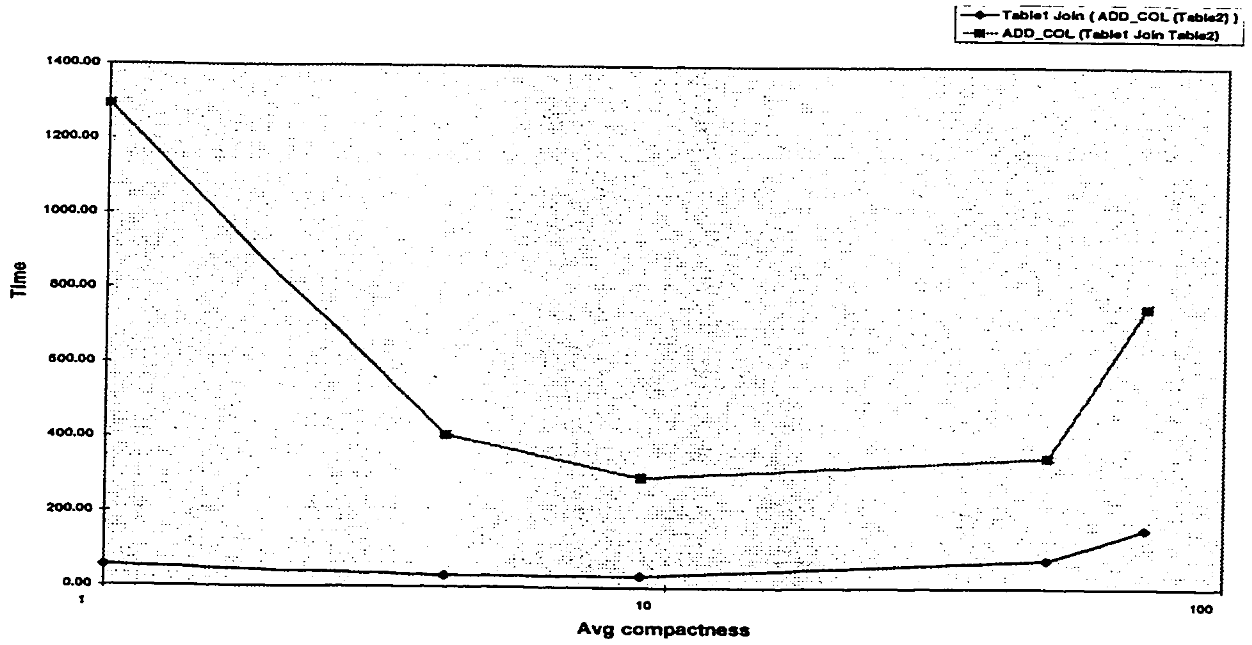

Figure 30: Performance of Join vs ADD_COL for varying avg compactness

Once more, these results are particularly interesting since they fall outside the bounds of our heuristics. Figure 30 shows how the compactness affects the efficiency of the ADD_COL operation. The behaviour corresponds to the following intuition: As the average compactness increases, the number of Merge Classes decreases, but for each Merge Class the number of tuples to be merged, and the width of the merged tuples, increases. The algorithm we used to implement the ADD_COL operator has the following features: (i) an overhead associated with each Merge Class, and (ii) a cost to merge tuples in a Merge Class which increases with the width of the merged tuples. Thus, a great number of small Merge Classes as well as a small number of very large Merge Classes are two extremes which are treated less efficiently than a medium number of Merge Classes each of medium size. Other algorithms could present different behaviours. Depending on the specific behaviour, it could become very important for the optimiser to use all available statistics about the data distribution in order to predict the efficiency of the operator.

The speed-up seems to follow a behaviour similar, by and large, to the one described above, and for similar reasons.

Chapter 7

Comparison With Other Work

In this chapter, the contributions of the thesis are compared with previous work in the field. Section 7.1 compares nD-SQL with previously proposed extensions to SQL, including SchemaSQL (Section 7.1.1), while Section 7.2 compares nD-SQL to related work on multi-database query optimisation.

7.1 SQL Extensions

There have been numerous extensions to SQL-like languages over the years, some inspired by multi-database interoperability requirements ([Lit89, GLRS93, SSR94, MR95]) and some motivated by querying OODBs ([KKS92, ASD+91, CL93]). Unlike nD-SQL, though, none of the above languages has both restructuring and complex aggregation capabilities.

First, Litwin et al. [Lit89, GLRS93] proposed a language called MSQL, capable of expressing multi-database queries "joining" data in different heterogeneous databases in one shot. [MR95] extends MSQL with external functions (for resolving semantic heterogeneity). MSQL (with this extension) does not treat schema and data in a uniform manner. So, schema independent querying and overcoming schematic heterogeneity pose a problem. UniSQL/M [KGK+95] is a relational multi-database system. However, SQL/M, the language supported by UniSQL/M, has limited support for manipulating meta-data. Thus, restructuring transformations and complex aggregations of the form supported by nD-SQL are hard to express in such an environment.

The idea of using of a catalog database in order to model relational data with the federation model has some similarities to the ideas used in the C-SQL project [SSR94] in that both are non-intrusive additions to existing database systems. However, unlike

nD-SQL, no formal semantics for C-SQL has been given and C-SQL does not have restructuring and complex aggregation capabilities.

Important extensions to SQL inspired by OODB querying include Kifer et al.'s XSQL [KKS92], Ahmed at al.'s HOSQL [ASD+91], and Chomicki and Litwin's OSQL [CL93]. XSQL permits very complex and powerful queries. However, the concern about its effective and efficient implementability has not been addressed by its authors. Both HOSQL and OSQL do not allow ad hoc queries that refer to more than one component database in one shot. Finally, it is not clear that the semantics of HOSQL and OSQL are downward compatible with SQL, and we are are not aware of any formal semantics for XSQL. The powerful emerging standard for SQL3 ([SQL96, Bee93]) supports ADTs, oid's, and external functions, but to our knowledge, does not *directly* support the kind of higher-order features for meta-data manipulation as in nD-SQL; programming such features would thus be very low level and tedious. Some of the expressions for extracting domain values and values of criteria in nD-SQL resemble the path expressions of OQL [Cat96]. However, path expressions are unlimited in their depth of nesting, whereas in nD-SQL, the design and semantics being inspired by multi-database interoperability, there is a natural bound on the levels of nesting for such expressions. There also seems to be no direct facility for restructuring in OQL.

Two noteworthy extensions to SQL from the vendor side are DB2/SQL [DB296] and ORACLE/SQL [ORA]. Of these, DB2/SQL is being incorporated in DataJoiner, IBM's new middleware for interoperability, and supports queries involving joins of tables from multiple DBMS in one select statement. As far as we know, restructuring and complex forms of aggregation of the kind supported in nD-SQL are not directly supported at a high level. ORACLE/SQL's DECODE feature is worth noting, since it permits some limited form of cross-tabbing. Still, this is very limited compared to the restructuring capabilities of nD-SQL.

Finally, Ross [Ros92] and Gyssens et al. [GLS96] are two recently proposed algebras which have the power of manipulating meta-data. Of these, the algebra in [Ros92] has limited restructuring capabilities, while that in [GLS96] has been shown to be complete for all generic restructuring transformations. However, both languages do not handle aggregation. Ross et al. [SRC97] generalise CUBE into a multi-feature CUBE, and propose fast algorithms for computing queries involving this operator. Their contributions and those of this thesis are complementary.

7.1.1 SchemaSQL

SchemaSQL is a multi-database interoperable query language proposed by Lakshmanan et al. [LSS96], capable of restructuring and complex aggregations, and is the closest language to nD-SQL. In particular, the syntax for database, relation, and column variables was inspired by SchemaSQL. However, there are the following major differences between the two languages.

1. *Lack of typing*: SchemaSQL offers no aids to the programmer to control an indiscriminate use of column/relation variables. This can lead to "ill-typed" and meaningless queries; e.g., it is easy to write a query in SchemaSQL that puts all values appearing in *all* columns of bse::prices into one output column! In the presence of aggregation, the problem gets even more serious.

2. *Limited restructuring*: At most one attribute domain can be placed in the relation/column dimension; e.g., one cannot transform the data in tse::quotes to the representation similar to bse::prices. Thus it is impossible to add or remove more than one criteria to or from a column at a time, nor is it possible to do cascaded additions or removals. Besides, unlike nD-SQL, only views, and not queries, can express restructuring, leading to an unpleasant asymmetry.

3. *Loss of meta-data*: The underlying model of SchemaSQL cannot keep track of meta-data against restructuring; e.g., when nyse::prices is restructured into the schema of mse, the fact that 'ibm' is a Ticker is lost. In nD-SQL, the notions of concepts and criteria are rich enough to always retain meta-data.

4. *Limited sub-aggregation*: SchemaSQL does not allow many sub-aggregates; e.g., it is impossible to compute the daily total price (over all stocks) for each measure type in bse::prices. By contrast, this is straightforward in nD-SQL (e.g., see query (Q3), page 19).

5. *Multiple granularity*: One of the strengths of nD-SQL is its ability to express multiple granularity aggregation, possibly together with multiple visualisations (see Section 3.2), something SchemaSQL cannot do. On the query processing side, unlike [LSS96], this thesis proposes an algebra and exploits its properties for query optimisation purposes.

7.2 Multi-database Query Optimisation

Much work has been done in the context of multi-database query optimisation, particularly in integrating data sources with diverse capabilities. See Haas et al. [Haa97] for a survey. Du et al. [DKS92], Qian [Qia96] and Florescu et al. [Flo95] are related works studying query optimisation in multi-database systems. The query optimisation concerns of this thesis are different: the focus is on algebraic optimisation of queries across multiple relational databases with heterogeneous schemas, where queries can involve attribute/value conflicts, restructuring, and complex OLAP-style aggregation. To my knowledge, optimisation in such a setting is new. There are many interesting open research problems in this context, which are currently under investigation.

In recent work, Ross et al. [CR96] propose syntactic constructs for expressing aggregations with multiple features. In [SRC97], they show how these constructs can be combined with the `CUBE` operator leading to multi-feature cube queries. The main contribution of these papers are: (i) the syntactic extensions to `SQL`, and (ii) an algorithm which translates these constructs into a query execution plan that minimises database scans. Thus, they generalise the data cube operator of Gray et al. [Gray+96]. As shown in Chapter 3, n`D-SQL` can express not only the `CUBE`, but many interesting and practically useful variations of it, together with multiple visualisations. A careful examination of [CR96, SRC97] reveals that the extensions to the `CUBE` proposed by them are orthogonal to those expressible in n`D-SQL`.

Chapter 8

Summary and Future Work

In this Chapter, we review the objective and contributions of this thesis (Section 8.1). We then identify and discuss the various avenues of future research opened up by this thesis (Section 8.2).

8.1 Summary

The dual problem studied by this thesis was to solve the schematic heterogeneity problem for interoperability among relational sources, while enabling OLAP-style computations to be performed on that data *in a non-intrusive fashion.*

We proposed the Federation Model, a formal model for a federation of relational sources with possibly heterogeneous schemas, which: (i) captures the diversity of schemas arising in practice, allowing a symmetric treatment of data and schema, and (ii) that captures the complete space of dimensional representations of data, fully exploiting the three physical dimensions implicit in the relational model. The notions of concepts, criteria and federated names, and the use of a catalog database to model existing relational sources, are central to the Federation Model.

We also proposed a query language called nD-SQL which makes use of the Federation Model and is capable of: (a) resolving schematic discrepancies among a collection of relational databases or data marts with heterogeneous schemas, and (b) supporting a whole range of multiple granularity aggregation queries like CUBE, ROLLUP, and DRILLDOWN, but, to an arbitrary, user controlled, level of resolution. In addition, nD-SQL can express queries that restructure data conforming to any particular dimensional representation into any other. We presented the syntax of nD-SQL and its semantics, which is downward compatible with the semantics of SQL.

The thesis also proposed an extension to relational algebra, capable of restructuring, called *restructuring relational algebra* (RRA). RRA is used as a vehicle for efficient processing of nD-SQL queries. We proposed an architecture for this purpose and we developed query optimisation strategies based on the properties of RRA operators. We have implemented the operators of the RRA and we have tested the performance of heuristics developed for query optimisation.

The non-intrusiveness of the nD-SQL Server architecture makes it an attractive option when there is a need for integration and OLAP-style analysis, of information locked away in individual autonomous sources which are hard to port to a common database, either because of security reasons, or because of the amount of labour involved. An nD-SQL Server can also be used during the life-cycle of a data warehouse, to interoperate between the various components that have not yet been fully integrated, while immediately providing a decision support mechanism.

8.2 Future Work

Many opportunities revealed by this thesis remain to be fully explored:

- **Semantics and Well-Formed queries:** We have been considering the impact of modifying the semantics in order to relax some conditions of well-formed queries. This would permit a richer class of queries to be expressible in nD-SQL. For example, a query like:

  ```
  SELECT   C.Year
  FROM     -> D, D -> R, D::R -> C
  WHERE    C HASA Year
  ```

  is not well-formed if the relations R ranges over do not all have the same schema. But it would be possible to modify the semantics such that, if each relation R ranges over has a complex column with criteria Year, the semantics would be well-defined;

- **Query Optimisation:** As mentioned in Section 5.1.3, several possibilities for further research exist in the nD-SQL context. First, additional interleaving of operators could be identified. Then, we should study the impact of using the notion of envelopes in efficient processing of remote queries dispatched by the nD-SQL server. In a recent paper ([SV98]), Subramanian and Venkataraman proposed the notion of *transient views* applicable for query optimisation in a single database. It is interesting to note that transient views are actually a special case of our notion of envelopes

The optimisation of nD-SQL queries with dimension variables is a very challenging area of research. Arbitrary sets of group-bys may need to be computed, while multiple renderings of the same result may need to be presented.

Efficient processing of the group-bys requires computing them in batch. In this context, connection of our work with Ross et al.'s multi-feature cubes ([SRC97]) merits a serious study. Exploiting materialised views for answering queries involving arbitrary group-bys is a promising area. An important piece of related work is [HRU96].

We also need to investigate how the coupling of a rendering engine with the optimisation engine would influence the efficient computation of nD-SQL queries with dimension variables.

- nD-SQL **Server Implementation:** The tests we have conducted merely scratched the surface. Thorough testing of the efficiency of the restructuring operators with respect to all the parameters identified in section 6.2.1 is mandatory and should lead to interesting discoveries.

  A few words need to be said about the efficient implementation of the RRA operators. For a commercial implementation of an nD-SQL Server, where performance is paramount, the RRA executor should be completely integrated with a database engine. The operators need to be coded as an integral part of the engine, at a low level. Such an approach was out of the scope of this thesis work. Instead, an existing database engine was used to store and interact with all tables (including catalog tables and VITs). Thus, the RRA operators were implemented as an application and hence were coded at a much higher level than the classical operators. They make use of the APIs provided by the existing database engine in order to, for example, add entries to the catalog tables, or scan a table to remove criteria from complex columns. Therefore, the threshold above which a given heuristic becomes useful would very likely be lower than the one reported by the tests (i.e. the query rewrite heuristics would be applicable even more often!) if the restructuring operators were coded at a lower level and integrated in the engine. We would like to efficiently code our operators at a much lower level in a query engine before completing our testing.

These, and others, are part of our ongoing work.

Bibliography

[ACM90] ACM *ACM Computing Surveys*, 22(3), Sept 1990. Special issue on HDBS

[ACM94] ACM *ACM Transactions on Database Systems*, Volume 19, June 1994

[Agar+96] Agarwal, S. *et al.* On the Computation of Multidimensional Aggregates In *Proc. 22nd VLDB Conf.*, 1996.

[Andr+96] Alanoly Andrews, Laks V.S. Lakshmanan, Nematollaah Shiri and Iyer N. Subramanian On Implementing SchemaLog: An Advanced Database Programming Language In *Proc. Intl. Conf. on Information and Knowledge Management*, Baltimore, MD, November 1996.

[ASD+91] Ahmed, R., Smedt, P., Du, W., Kent, W., Ketabchi, A., and Litwin, W. The Pegasus Heterogeneous Multidatabase System. *IEEE Computer*, December 1991.

[Bee93] Beech, D. Collections of Objects in SQL3. In *Proc. 19th VLDB Conf.*, 1993.

[Bee87] Beeri, Catriel and Ramakrishnan, Raghu On the Power of Magic In *Proc. PODS 1987*

[Cat96] Cattell, R.G.G. *The Object Database Standard: ODMG-93 Release 1.2.* Morgan-Kauffmann, San Francisco, CA, 1996.

[CD97] Chaudhuri, Surajit and Dayal, Umesh. An Overview of Data Warehousing and OLAP Technology, Tutorial - VLDB'96 and SIGMOD'97, SIGMOD Record '97.

[CL93] Chomicki, J. and Litwin, W. Declarative Definition of Object-Oriented Multidatabase Mappings. In Ozsu, M.T, Dayal, U, and Valduriez, P, editors, *Distributed Object Management.* M. Kaufmann Publishers, Los Altos, California, 1993.

[Dat95] IBM DataJoiner - A Multidatabase Server Second Edition, May 1995 http://www.software.ibm.com/data/pubs/papers/dj1wp.ps

[DB296] *IBM DB2 for MVS/ESA Version 5* , 1996. – Programmer's Manual.

[DKS92] Du, Weimin, Krishnamurthy, Ravi, and Shan, Ming-Chien. Query Optimization in a Heterogeneous DBMS. In *Proc. Int. Conf. on Very Large Data Bases.*, pages 277–291, Dublin, Ireland, 1992.

[CR96] Chatziantoniou, Damianos and Ross, Kenneth A.. Querying Multiple Features of Groups in Relational Databases. In *Proc. 22th VLDB Conf.*, pages 295–306, Bombay, India, September 1996.

[Flo95] Florescu, Daniela, Rachid, Louiqa and Valduriez, Patrick Using Heterogeneous Equivalences for Query Rewriting in Multidatabase Systems. In *Proc. 23rd Int. Conf. on Cooperative Information Systems*, 1995.

[GL97] Gyssens, Marc and Lakshmanan, Laks V.S.. A Foundation for Multi-Dimensional Databases. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, pages 106–115, Athens, Greece, August 1997.

[GL98] Gingras, Frédéric and Lakshmanan, Laks V.S. *n*D-SQL: A Multi-dimensional Language for Interoprability and OLAP. To appear in *Proceedings of the 24th Int. Conf. on Very Large Data Bases*, New York, USA, August 24-27, 1998.

[GLRS93] Grant, J., Litwin, W., Roussopoulos, N., and Sellis, T. Query Languages for Relational Multidatabases. *VLDB Journal*, 2(2):153–171, 1993.

[GLS96] Gyssens, Marc, Lakshmanan, Laks V.S., and Subramanian, Iyer N. Tables as a Paradigm for Querying and Restructuring. In *Proc. ACM Symposium on Principles of Database Systems (PODS)*, June 1996.

[GLS$^+$97] Gingras, Frédéric, Lakshmanan, Laks V.S., Subramanian, Iyer N., Papoulis, Despina, and Shiri, Nematollaah. Languages for Multi-database Interoperability. In *Proc. of the ACM SIGMOD*, Tucson, Arizona, May 1997. Tools Demo.

[Gray+96] Gray, J. and Bosworth, A. and Layman, A. and Pirahesh H.. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals In *Proc. of the 12th Intl. Conf. on Data Engineering (ICDE)*, 1996.

[Haa97] Haas, Laura *et al.* Optimizing Queries Across Diverse Data Sources. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, pages 276–285, Athens, Greece, August 1997.

[HRU96] Harinarayan, V., Rajaraman, A. and Ullman, J.D.. Implementing data cubes efficiently. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 205–216, 1996.

[KCGS93] Kim, W., Choi, I., Gala, S. K. and Scheevel, M. On resolving schematic heterogeneity in multidatabase systems. In *Distributed and Parallel Databases*, 1(3), 1993.

[KGK$^+$95] Kelley, W., Gala, S. K., Kim, W., Reyes, T.C., and Graham, B. Schema Architecture of the UniSQL/M Multidatabase System. In *Modern Database Systems*, 1995.

[KLK91] Krishnamurthy, R., Litwin, W., and Kent, W. Language Features for Interoperability of Databases With Schematic Discrepancies. In *ACM SIGMOD Intl. Conference on Management of Data*, pages 40–49, 1991.

[KKS92] Kifer, Michael, Kim, Won, and Sagiv, Yehoshua. Querying Object-Oriented Databases. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 393–402, 1992.

[Lit89] Litwin, W. MSQL: A Multidatabase Language. *Information Science*, 48(2), 1989.

[LR9289] Landers, T. and Rosenberg, R An overview of multibase *Distributed Databases*, pages 153-184, 1982

[LSS96] Lakshmanan, L.V.S., Sadri, F., and Subramanian, I. N. SchemaSQL – a Language for Querying and Restructuring multidatabase systems. In *Proc. IEEE Int. Conf. on Very Large Databases (VLDB'96)*, pages 239–250, Bombay, India, September 1996.

[MR95] Missier, P. and Rusinkiewicz, Marek. Extending a Multidatabase Manipulation Language to Resolve Schema and Data Conflicts. In *Proc. Sixth IFIP TC-2 Working Conf. on Data Semantics (DS-6)*, Atlanta, May 1995.

[ORA] *Oracle7 Server SQL Reference.* available from: http://www.oracle.com/documentation/sales/html/o7sqlref.html.

[Qia96] Quian, Xiaolei. Query Folding. In *Proc. IEEE Int. Conf. on Data Eng.*, New Orleans, LA, February 1996.

[Ros92] Ross, Kenneth. Relations With Relation Names as Arguments: Algebra and Calculus. In *Proc. 11th ACM Symp. on PODS*, pages 346–353, June 1992.

[S97] Subramanian, Narayana Iyer A Foundation for Integrating Heterogeneous Data Sources Ph.D. Thesis, Concordia University, Montreal, Quebec, August 1997.

[SQL96] SQL Standards Home Page. SQL 3 articles and publications, 1996. URL: www.jcc.com/sql_articles.html.

[SRC97] Srivastava, Divesh Ross, Kenneth A. and Chatziantoniou, Damianos. Complex Aggregation at Multiple Granularities. In *Proc. 23rd Int. Conf. on Very Large Data Bases*, pages 116–125, Athens, Greece, August 1997.

[SSR94] Sciore, E., Siegel, M., and Rosenthal, A. Using Semantic Values to Facilitate Interoperability Among Heterogeneous Information Systems. *ACM Transactions on Database Systems*, 19(2):254–290, June 1994.

[SV98] Subramanian, Subbu N. and Venkataraman, Shivakumar Cost-Based Optimization of Decision Support Queries Using "Transient Views" in *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pp.319-330, Seattle, Washington, June 2-4, 1998.

[Tem87] Templeton, M. Mermaid: A front-end to distributed heterogeneous databases In *Proc. IEEE 75, 5*, pages 695-708, May 1987.

[ZDN97] Zhao, Yihong, Deshpande, Prasad M., and Naughton, Jeffrey F.. An Array-Based Algorithm for Simultaneous Multidimensional Aggregates In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 159–169, Tucson, Arizona, 1997.

Appendix: Grammar of nD-SQL

```
nD-SQLQuery := select_clause
               from_clause
               where_clause
               [groupby_clause
                [having_clause] ]

select_clause := SELECT select_list
                 | (SELECT select_list) [AS label] FOR crit_list

select_list := select_object {, select_object}

select_object := domain [AS label] [FOR crit_list]
                 | agg_op(domain) [AS label] [FOR crit_list]

domain := db_var
          | tuple_var.attribute
          | tuple_var.col_var
          | col_var.criterion
          | rel_var.criterion

agg_op := SUM
          | COUNT
          | MAX
          | MIN
          | AVG

label := label_piece { & label_piece}

label_piece := domain
               | "label_string"

crit_list := domain {, domain}

from_clause := var_dec {, var_dec}

var_dec := -> db_var
         | db_var -> rel_var
         | db_var :: rel_var tuple_var
         | db_var :: rel_var -> col_var

db_var := string

rel_var := string

tuple_var := string

col_var := string

attribute := string

criterion := string

concept := string

where_clause := where_cond { AND | OR where_clause}
                | (where_cond { AND | OR where_clause})
                | where_cond { AND | OR (where_clause)}
                | (where_cond { AND | OR (where_clause)})
```

```
where_cond := hasa_cond
            | isa_cond
            | rel_op_cond
            | other_cond

hasa_cond := rel_var HASA criterion
           | col_var HASA criterion

isa_cond := rel_var ISA concept
          | col_var ISA concept

rel_op_cond := operand rel_op operand

operand := domain
         | value

value := any_string

string := character {character}

character := one of the characters among a-zA-Z0-9_

label_string := label_character {label_character}

label_character := one of the characters among a-zA-Z0-9.;:/\+=
                                               -_!@#$%^&*()~|<>

any_string := any_character {any_character}

any_character := any character from the ASCII set that is acceptable
                 in values in a tuple

rel_op := =
        | <>
        | >
        | <
        | <=
        | >=

other_cond := other type of conditions allowed in any of the flavors of SQL.
              where operands can be domains when appropriate.
              E.g.: LIKE conditions, IN conditions, EXISTS condition

groupby_clause := domain {, domain}

having_clause := having_cond { AND | OR having_clause}
               | (having_cond { AND | OR having_clause})
               | having_cond { AND | OR (having_clause)}
               | (having_cond { AND | OR (having_clause)})

having_cond := having_operand rel_op having_operand

having_operand := domain
                | agg_op(domain)
                | value
```