nD-SQL: A Multi-dimensional Language for Interoperability and OLAP

Frédéric Gingras and Laks V.S. Lakshmanan
Department of Computer Science, Concordia University,
Montréal, Québec H3G 1M8
e-mail: {gingras, laks}@cs.concordia.ca

Abstract

We propose a multi-dimensional language called nD-SQL with the following features: (i) nD-SQL supports queries that interoperate amongst multiple relational sources with heterogeneous schemas, including RDBMS and relational data marts, overcoming the mismatch between data and schema; (ii) it supports complex forms of restructuring that permit the visualization of n-dimensional data using the three physical dimensions of the relational model, viz., row, column, and relation; (iii) it captures sophisticated aggregations involving multiple granularities, to an arbitrary degree of resolution compared to CUBE, ROLLUP, and DRILLDOWN. We propose a formal model for a federation of relational sources and illustrate nD-SQL against it. We propose an extension to relational algebra, called restructuring relational algebra (RRA), capable of restructuring and aggregation. We propose an architecture for the implementation of an nD-SQL server, based on translating nD-SQL queries into equivalent RRA expressions, which are then optimized. We are currently implementing an nD-SQL server on the PC platform based on these ideas.

1 Introduction

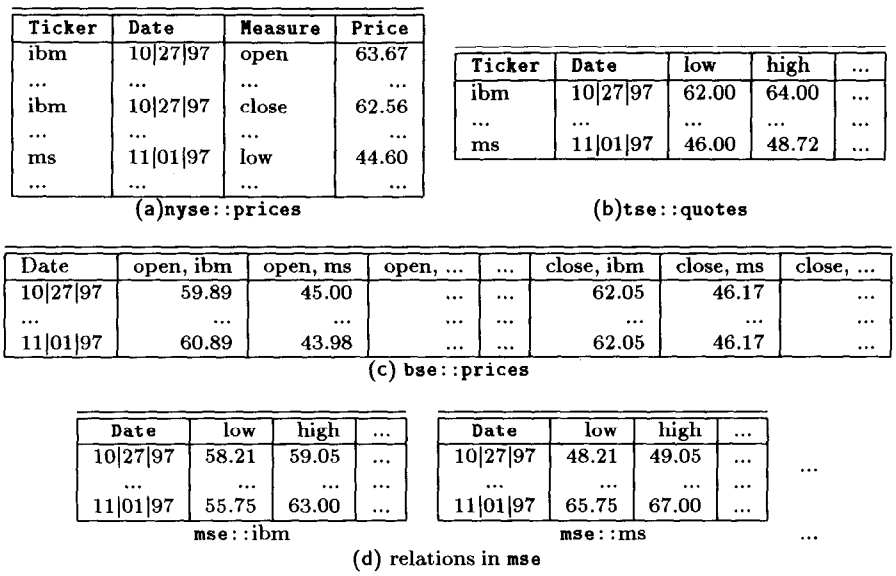
Interoperation among multiple heterogeneous databases continues to be an important practical problem. It entails resolving incompatibilities and conflicts between component database systems on a number of different fronts, including platforms, database schemas, and transaction management systems, to name a few. The importance of developing query languages capable of "cross-querying" the component databases, overcoming the discrepancies among their schema and data semantics has been recognized (see [CL93, GLRS93, KLK91, Lit89, SSR94, LSS96] for a few proposals for such languages). It has been realized from these earlier works that even in the context of a federation consisting of relational databases, the conflicts among the component database 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). It is known that conventional languages like SQL or variants cannot be used to overcome this conflict (see [LSS96]), without a host language.

In this paper, we view interoperability in a slightly larger context where the objective is not only to run traditional SQL queries on the data in a federation, but also queries involving multiple granularity aggregation required for OLAP. Typically, such queries involve operators like CUBE, ROLLUP, and DRILLDOWN. More precisely, the problem studied in this paper is: *how to develop a query language compatible with* SQL, *that is capable of (i) expressing queries on a federation of relational sources resolving the conflicts between the component schemas, and (ii) expressing OLAP queries involving multiple granularity aggregations?*

The motivation for the above problem is as follows. First, consider a complex organization whose data is distributed among its functional or departmental units. Decision support requires: (i) interoperability among component databases, and (ii) eventually the creation of a data warehouse storing integrated summaries of the operational data, providing efficient support for OLAP queries.[1] *Once* a data warehouse is created, the discrepancies among component databases are resolved and the data would be integrated. Why then need yet another query language? However, as discussed in [CD97], building a data warehouse is

Proceedings of the 24th VLDB Conference
New York, USA, 1998

[1] Actually, a data warehouse should ideally support both OLAP and mining as argued by Chaudhuri and Dayal [CD97], but in this paper, we do not consider mining.



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

(a)nyse::prices

Ticker	Date	low	high	...
ibm	10|27|97	62.00	64.00	...
...	...	...	...	...
ms	11|01|97	46.00	48.72	...

(b)tse::quotes

Date	open, ibm	open, ms	open, ...	...	close, ibm	close, ms	close, ...
10|27|97	59.89	45.00	...	...	62.05	46.17	...
...	...	...	...	...	...	...	...
11|01|97	60.89	43.98	...	...	62.05	46.17	...

(c) bse::prices

Date	low	high	...
10|27|97	58.21	59.05	...
...	...	...	...
11|01|97	55.75	63.00	...

mse::ibm

Date	low	high	...
10|27|97	48.21	49.05	...
...	...	...	...
11|01|97	65.75	67.00	...

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`.

a long, complex, and expensive process, often taking up to several years to complete. Many organizations 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. *Thus, in the evolutionary lifecycle 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 queries.

Next, consider interoperation of a general federation of databases, not necessarily belonging to any one organization. The participants of the federation may not permit the data in their databases to be integrated into a central warehouse. One approach that has been followed in the past to resolve schematic discrepancies is to convert the data in the databases to conform to a common canonical schema, by defining mappings (e.g., see [ASD+91]). Unfortunately, such mappings tend to be very low level and converting data in this manner is labor intensive, necessitating lengthy and costly human interventions. This once again calls for a high level query language capable of resolving such conflicts automatically, assuming additional information on the component schemas is added to the federation in a non-intrusive manner. In this context, even though traditionally interoperability has been posed as a problem without the requirement to support OLAP queries, we anticipate there are many applications which can benefit from such a feature. For example, in a stock market federation, an investment broker or analyst might wish to compute multiple granularity summaries on the data pooled from a number of exchanges, in order to study the performance of stocks and funds.

In this paper, we propose a formal model for a federation of relational databases with possibly heterogeneous schemas (Section 2). We also propose an n-dimensional query language called nD-SQL, 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 to any other (Section 3). We propose 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 propose an architecture for this purpose. We develop query optimization strategies based on properties of RRA operators (Section 4). We also discuss the implementation of a system based on our ideas (Section 5). We finally compare nD-SQL and its approach with related work (Section 6).

Before concluding this section, we briefly illustrate the power of nD-SQL. In nD-SQL, it is straightforward to restructure the data in any of the databases in Figure 1, to the schema of any other database, which is impossible in most known query languages, without external calls to procedures in a host language. For lack of space, we suppress the proofs of all our results and finer details of our query processing algorithms in this paper, and refer the reader to the full version [GL98]. We also point the reader to the URL http://www.cs.concordia.ca/~special/bibdb/nd-sql for more information about nD-SQL.

2 The Model

In this section, we propose a formal model for collections of relational databases. The highlights of this model are: (i) It captures heterogeneous schemas of relational databases arising in practice, treating data and schema symmetrically; (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 section.

We begin with the notion of a scheme. The size of practical database schemas, may be data dependent (e.g., the number of columns of `tse` and the number of relations in `mse`, in Figure 1), unlike in the classical relational model. Our notion of a "federation scheme", proposed next, makes it possible to elegantly 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. We assume pairwise disjoint, infinite, sets of names, $\mathcal{N}$, values, $\mathcal{V}$, and id's, $\mathcal{O}$. We use typewriter font 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 (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., $X \neq \emptyset$). We usually denote simple federated names $(N, \emptyset)$ 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.*

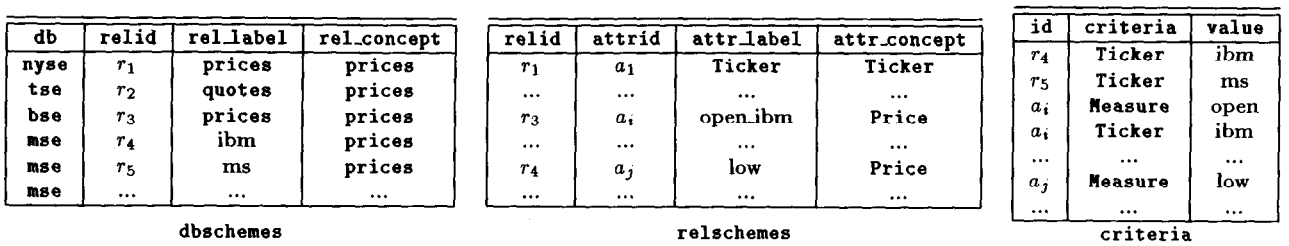
The intuition behind the above definition is twofold: (1) A complex attribute (resp., relation) name translates to a *set* of complex column (resp., relation) labels in an instance. For example, the complex attribute name (`Price`, {`Measure`, `Ticker`}) in the scheme might correspond in an instance to the set {`Price FOR Measure` = *low* AND `Ticker` = *ibm*, ..., `Price FOR Measure` = *close* AND `Ticker` = *hp*} of column labels. The federation scheme of the instance shown in Figure 1 is: $\mathcal{S}_1$ = {`nyse::prices(Ticker, Date, Measure, Price)`, `tse::quotes(Ticker, Date, (Price, {Measure}))`, `bse::prices(Date, (Price, {Measure, Ticker}))`, `mse::(prices, {Ticker})(Date, (Price, {Measure}))`}. 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". We will return to this point later. (2) The notion of a federated relation scheme formalizes 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})`), domain values of relation criteria (`Ticker`) are placed along the relation dimension, domain values of criteria of complex columns (`Measure`) along the row dimension, and domain values of simple columns (`Date`) along the column dimension.

Definition 2.2 (Federation Instance) *Let $\mathcal{S} = \{\mathtt{d}_1 :: R_1(C_1, \ldots, C_k), \ldots, \mathtt{d}_m :: R_m(D_1, \ldots, D_n)\}$, the $\mathtt{d}_i$ 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 $\mathcal{S}$.*
- *$rel : \mathcal{D} \rightarrow 2^{\mathcal{O}}$ is a function that maps each database name in $\mathcal{D}$ to a finite set of relation id's. Below, we will use $\mathcal{R} = \bigcup_{d \in \mathcal{D}} rel(d)$ to denote the set of all relation id's in the instance.*
- *$col : \mathcal{R} \rightarrow 2^{\mathcal{O}}$ 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 $col(r)$.*
- *$conc : \mathcal{O} \rightarrow \mathcal{N}$ is a function that maps each id to a name, called its underlying concept.*
- *$crit : \mathcal{O} \rightarrow 2^{\mathcal{N}}$ 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. The concepts and criteria associated with these labels are typically *not* recorded in real-life federations. However, *intuitively*, we can understand that the concept associated with the label "low" is `Price` and that the only associated criterion is `Measure`. In the sequel, we shall refer to the formal notion of instances defined above 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 2.3 (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.*



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.

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. $crit(a) \neq \emptyset \neq crit(b)$, we require 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 $a \in col(r)$, we require $t[a] \in dom(conc(a))$, 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 $N \in crit(a)$, we require $val(a, N) \in dom(N)$, i.e. the values associated with criteria should belong to the appropriate domains.*

In the sequel, when we refer to abstract instances, we mean legal (abstract) instances.

Real Federations and Formal Model Bridged: Two questions need to be addressed now: (1) How can real-life federations be captured in the formal framework? (2) How relevant is our formal notion of abstract federation instances to practice, and specifically, for the purpose of interoperability? To deal with question 1, we define real instances.

Definition 2.4 (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. We call such a real instance $\mathcal{F}$ 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 2.2. 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, we may have no 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.

We next address question 2 above. We can connect abstract and real instances 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, formalized next.

Definition 2.5 (Catalog Database) *The catalog database associated with an abstract instance $\mathcal{I}$ consists of the following three relations (which we call 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.

We treat the database `catalog` as a distinguished database from a formal viewpoint in that it always consists of the three catalog tables defined above. *We stress that casual users do not have to explicitly manipulate the catalog db.* For linking an abstract instance to its corresponding real instance, we propose the notion of an augmented instance. Let $\mathcal{F}$ be a real instance corresponding to an abstract instance $\mathcal{I}$. The *augmented instance* associated with $\mathcal{F}$ and $\mathcal{I}$ is the federation obtained by adding to $\mathcal{F}$ the distinguished database `catalog`, the catalog database associated with $\mathcal{I}$. We then have the following theorem:

Theorem 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.* ∎

Incorporating data marts: So far, we have focused attention 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. We 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 and 2.2 subsume relational data marts.

3 Syntax and Semantics of *n*D-SQL

In this section, we present the syntax of *n*D-SQL by explaining the additions made to SQL. The semantics of *n*D-SQL will be illustrated with examples. The complete syntax of the language and a rigorous account of the semantics can be found in [GL98]. Tables summarizing the differences in syntax between SQL and *n*D-SQL are available at http://www.cs.concordia.ca/~special/bibdb/ndsql. *Throughout, we will use the federation of Figure 1 as a running example to illustrate our queries.*

3.1 Multi-dimensionality and Restructuring

*n*D-SQL uses the classic SELECT, FROM, WHERE, GROUP BY and HAVING clauses of SQL, and adds to the syntax in several manners. (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).* (2) WHERE clause: We introduce two new interpreted constraints 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 we will formally define at the end of the present section.
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
```

Here, R *is a rel_var restricted to range over the relations of database* mse *having* Ticker *values as criteria values, and* C *is a clmn_var restricted to range over the columns of these relations having* Price *values as their underlying concept.*
(3)SQL has a unique kind of domain expression, tuple_var.attr (abbreviated as attr). In addition to this, *n*D-SQL also has the domain expressions tuple_var.clmn_var and C.criterion, where C is a relation/column variable and criterion is one of the criteria of the relations/columns it ranges over. This expression serves to extract criteria values. All of these domain expressions can be used in the SELECT and GROUP BY clauses, and in conditions in the WHERE and HAVING clauses. We define the underlying concept of a domain as follows:

Definition 3.1 (Underlying concept of a domain)
undconc(domain) =

attribute	*if domain is of the form* tuple_var.attribute
criterion	*if domain is of the form* rel_var.criterion
criterion	*if domain is of the form* clmn_var.criterion
concept(clmn_var)	*if domain is of the form* tuple_var.clmn_var

where we refer to the concept of a complex column or relation over which a var ranges as concept(var). We will also refer in the sequel to the set of criteria of the same column or relation var 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, we need to deposit data values as criteria values. The syntax for depositing data values as column criteria values is to use the following new type of select_objects in the SELECT clause:
domain$_0$ [AS *label*] FOR (*domain*$_1$ {, *domain*$_i$}), $i > 1$
where the optional *label* can be any combination of constant strings concatenated (using the "&" symbol) with any combination of the domains *domain*$_j$, $j \geq 1$. Examples of labels could be: "Price for Year = "&T.Ticker, "Price for "&T.Ticker, T.Ticker&"'s Price" or even simply T.Ticker. When no label (AS subclause) is present, appropriate default conventions for labels are used [GL98].
The use of the FOR subclause with a select_object indicates that there should be a complex attribute with name (undconc(*domain*$_0$), {undconc(*domain*$_1$), undconc(*domain*$_2$), ...}) (see Definition 3.1) 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, T.Date,
(Q2)         T.Price AS T.Measure FOR T.Measure
      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 *subclause. Note also how these* Measure *values are used as column labels.*
(5)To deposit data values as relation criteria, we englobe all the select_objects of the SELECT clause in parentheses and apply the following additional FOR subclause:

SumPrice FOR Measure = open	SumPrice FOR Measure = close	...
6521K	5475K	...

output::10|27|97

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

output::11|01|97

Figure 3: Result of query Q3

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

which indicates that a relation with name (rel_k, {undconc($domain_1$), undconc($domain_2$), ...}) should be created. The relation concept rel_k is generated by the system 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   (Sum(T.C) AS "SumPrice 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 *subclause restructures the sums into multiple columns, one per value of* Measure*, while the outer* FOR *subclause restructures the result into multiple relations, one per value of* Date*. The result of the query is shown in Figure 3, where we assume all output relations to be temporarily viewed as members of a database named "output".*

Various abbreviations are acceptable in *n*D-SQL syntax [GL98], whose details are suppressed for lack of space.

Well Typing: Intuitively, a query can be meaningful only if it maps legal instances to legal instances. More precisely, we have the following definition.

Definition 3.2 (Well-Typing) *A* nD-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. We develop such an algorithm below.

It turns out that there are simple rules that the user can follow in order to make sure a query is well-typed. In particular, let us call a query Q *well-formed*, provided it satisfies the following conditions.

- relation variables must be restricted (by ISA and HASA conditions) to range over relations having the same concept and criteria set;
- attribute variables must be restricted (by ISA and HASA conditions) to range over columns having same concept and same set of criteria;
- all the complex columns created in the SELECT clause have the same set of criteria;

The following is a syntactic characterization of well-typing.

Theorem 3.1 *A query is well-typed if and only if it is well-formed.*

Theorem 3.1 immediately yields an algorithm 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 [GL98].

3.2 Enhancing *n*D-SQL for OLAP: multiple visualizations and subaggregates

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 recognized [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, we contend that in general, depending on the application at hand, users may be interested in subsets that need not be covered by these operators (see Example 3.4 e.g.). In this section, we develop some simple extensions to *n*D-SQL and show how they lead to a powerful mechanism for expressing arbitrary subsets of group-bys. In addition, we will also show that together with the restructuring capabilities of *n*D-SQL, this allows us to compute arbitrary multiple granularity aggregations and visualize the results in multiple ways. Following OLAP terminology, we refer to each of the names in a federation scheme as a *logical dimension*. More precisely, we have the following

Definition 3.3 (Logical Dimensions) *The logical dimensions of a federated relation scheme $R(C_1, \ldots, C_n)$ are the set of concepts of C_i together with the set of criteria of R, and of the complex columns among C_i, $1 \leq i \leq n$. Let Q be an* nD-SQL *query and let $R_1, \ldots, R_m$ be the set of federated relation schemes mentioned in Q. Then the set of logical dimensions associated with Q is the union of the logical dimensions associated with R_i, $1 \leq i \leq m$.*

For example, the dimensions of each of the four federated relation schemes in $\mathcal{S}_1$, corresponding to the instance of Figure 1 are Ticker, Date, Measure, Price. We do not address the issue of dimension hierarchies in this paper.

The main enhancement to *n*D-SQL syntax is a new kind of variable called *dimension variable* (declared as DIM var), ranging over the *names* of all logical dimensions associated

with the query, except those being aggregated. An *n*D-SQL query Q with dimension variables is *equivalent* to a set of *n*D-SQL queries *without* dimension variables, obtained by instantiating the dimension variables in Q to all possible combinations of dimension names *that satisfy the constraints on the dimension variables*, specified in the WHERE clause of Q. We start with an extremely simple example to illustrate the ideas.

Example 3.1

```
     SELECT   X, SUM(T.Price)
(Q4) FROM     nyse::prices T, DIM X
     GROUP BY X
```

The only dimension variable is X. *The only federated relation scheme mentioned in* (Q4) *is* nyse::prices, *whose associated dimensions are* Date, Ticker, Measure, Price. *Of these,* Price *is being aggregated. So, the dimension variable* X *ranges over the dimension names* Date, Ticker, *and* Measure. *The equivalent set of queries without dimension variables are as follows.*

```
       SELECT   T.Ticker, SUM(T.Price)
(Q4a)  FROM     nyse::prices T
       GROUP BY T.Ticker

       SELECT   T.Date, SUM(T.Price)
(Q4b)  FROM     nyse::prices T
       GROUP BY T.Date

       SELECT   T.Measure, SUM(T.Price)
(Q4c)  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 – Ticker, Date, *and* Measure. ∎

Constraints on dimension variables include the standard rel-ops $=, \leq, <, >, \geq, \neq$. We interpret them w.r.t. the lexicographic ordering of the dimension names. E.g., Date < Ticker. We introduce a special constant, NONE, inspired by the special constant all introduced by Gray et al. [Gray+96].[2] We give this constant a special status w.r.t. the way the rel-ops are interpreted. *We assume: (i)* NONE *Op* NONE *is always true for all rel-ops Op; (ii)* ⟨*dimension*⟩ < NONE *is always true, for all dimension names* ⟨*dimension*⟩. Besides rel-ops, we also allow constraints involving the IN operator, with the obvious semantics. Finally, we introduce a special type of constraint using which we can allow a dimension variable to assume the value NONE. This feature is particularly useful for specifying multiple granularity aggregations, as our examples will show.

Example 3.2 *Let us now revisit the previous example and see how we can express a* CUBE *of* Price *values over the dimensions* T.Ticker, T.Date *and* T.Measure.

```
     SELECT   X, Y, Z, SUM(T.Price)
     FROM     nyse::prices T, DIM X,Y,Z
(Q5) WHERE    X < Y < Z AND DIMS CAN BE NONE
     GROUP BY X, Y, Z
```

[2] We simply find the name NONE more appropriate for the use we have for this constant here.

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. Finally, if we modify the constraints on dimension variables to:* X IN {T.Date, NONE} AND Y IN {T.Measure, NONE} AND Z IN {T.Ticker, NONE} AND X < Y < Z, *then this produces exactly the* group-bys {T.Date, T.Measure, T.Ticker}, {T.Date, T.Measure}, {T.Date}, *and* {}, *corresponding to* ROLLUP. ∎

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

Example 3.3

```
     SELECT   (AVG(T.Price) AS Y FOR Y) AS X FOR X
(Q6) FROM     nyse::prices T, DIM X, Y
     WHERE    DIMS IN {T.Date, T.Measure, T.Ticker}
     GROUP BY X, Y
```

This query generates all possible groupings of AVG(T.Price) *along two logical dimensions among* Date, Measure *and* Ticker. *Furthermore, it restructures each particular grouping in multiple ways along (physical) relation and row dimensions such that multiple visualizations of the same data are provided at once, as shown in Figure 4(b).* ∎

Our last example in this section illustrates the power of *n*D-SQL to generate sets of multiple granularity aggregations which do not seem to be obviously expressible using a combination of operators like CUBE, ROLLUP and/or DRILLDOWN.

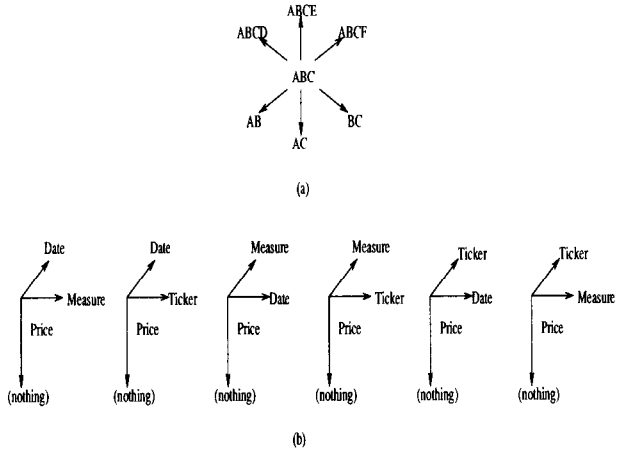


Figure 4: (a) "neighborhood" operator (b) Visualizations or result of query Q6

Example 3.4

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 want to look at the "neighborhood" 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}, {A,B}, {A,C}, *and* {B,C} *which form the neighborhood of* {A,B,C} *in the cube lattice. This query can be expressed as follows.*

```
        SELECT W, X, Y, Z, SUM(G)
(Q7)    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 4(a) depicts the "shape" of this set of group-bys. It is not clear how such a query can be expressed using known operators. ∎

4 Query Processing

We will discuss in this section our approach to an efficient implementation of the *n*D-SQL language. In order to simplify the presentation, we will first cover the processing of queries that do not involve dimension variables (Section 4.1). We will then discuss the processing of those queries involving dimension variables (Section 4.3).

4.1 Processing of queries that do not involve dimension variables

Overview: In order to efficiently process *n*D-SQL queries, we will define a new Restructuring Relational Algebra (RRA) which extends classical Relational Algebra (RA) with restructuring operators. Thus, to process *n*D-SQL queries we will translate them into equivalent RRA expressions, just like SQL queries are translated into RA expressions. We will then take advantage of the properties of the RRA operators to optimize the expressions. We can also take advantage of downward compatibility of RRA with RA to push some of the processing to remote databases. Our architecture is illustrated in Figure 6. Its highlights are that it is non-intrusive, requiring minimal extensions to existing technology, for deployment on top of existing SQL systems.

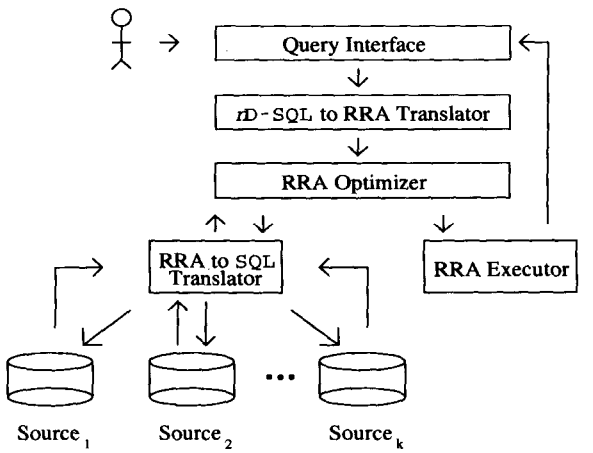


Figure 6: System Architecture

Restructuring Relational Algebra: RRA consists of the classical RA operators (that we extend slightly), 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 our model, simple columns of relations are denoted as in the classical relational model, while complex columns are of the form (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, we denote such complex columns as $(\text{concept}, t_{\text{criteria}})$, where t_{criteria} is the tuple that maps criteria to $\vec{v}$. We sometimes refer to t_{criteria} as a *criteria-tuple*. A similar remark applies for complex relations. The operators of RRA are thus: σ, Π, $\bowtie$, ADD_COL, REM_COL, ADD_REL and AGG where the latter can be any of the usual aggregation operators.

We first define the new operators, then explain how the classical ones are extended.

Definition 4.1 (Add Criteria to Columns) *The operation* $\text{ADD\_COL}_{critList \rightarrow concList}(\texttt{rel})$*, 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* $cols(r') = (cols(r) - \{C \mid C \text{ is a column of } r \text{ with concept in } critList\} - \{C \mid C \text{ is a column of } r \text{ with concept in } concList\}) \cup \{(C, t_{critList}) \mid C \text{ is a column of } r \text{ with concept in } concList\} \wedge \exists t \in r : t[critList] = t_{critList}\}$*. 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* $inst(r') = \{t \mid \forall (C, t_{critList}) \in cols(r') - cols(r) : \exists s \in r : \forall A \in cols(r) \cap cols(r') : t[A] = s[A] \wedge t_{critList} = s[critList] \wedge t[(C, t_{critList})] = s[C]\}$.

It should be noted that the column 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 5.

Definition 4.2 (Remove Criteria from Columns) *The operation* $\text{REM\_COL}_{critList}(\texttt{rel})$*, 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. 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* $cols(r') = \{A \mid A \text{ is a simple column in } cols(r)\} \cup \{(A, t_C)[\mathcal{C} - critList]) \mid (A, t_C) \text{ is a complex column in } cols(r)\} \cup critList$.
- *The instance of r′ consists of a set of tuples over* $cols(r')$*, defined as* $inst(r') = \{t \mid \exists s \in r : \exists \text{ a criteria-tuple } t_C : (\forall \text{ simple column } A \in cols(r) : t[A] = s[A]) \wedge (\forall \text{ complex column } (C, t_C) \in cols(r) : t[(C, t_C[\mathcal{C} - critList])] = s[(C, t_C)]) \wedge t[critList] = t_C[critList]\}$.

E.g., the expression $\text{REM\_COL}_{\text{Measure}}$ (ny2t :: prices), applied to the relation ny2t::prices of Figure 5, exactly yields the relation nyse::prices of Figure 1.

Ticker	Date	open	close	low	...
ibm	10|27|97	63.67	62.56	62.00	...
...	...	...	...	...	...
ms	11|01|97	47.02	46.50	44.60	...

Figure 5: `ny2t::prices`

Date	open	close	low	...
10|27|97	63.67	62.56	61.33	...
...	...	...	...	...
11|01|97	66.30	64.33	62.55	...

`ny2m::ibm`

Date	open	close	low	...
10|27|97	43.25	44.00	42.35	...
...	...	...	...	...
11|01|97	47.02	46.50	44.60	...

`ny2m::ms`

Figure 7: `ny2m::prices`

Definition 4.3 (Add Criteria to Relations) *The operation* ADD_REL$_{critList}$(`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* $t_{critList_i}$ *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 of the above operator, the expression ADD_REL$_{Ticker}$(`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 7.

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 our query processing algorithms. We point the reader to [GL98] for the details as well as for an algorithmic presentation of the restructuring operators.

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 to enumerating every column label and applying the same operation to each (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.

In general, operators of our RRA commute provided certain conditions are met.

Theorem 4.1 *Commutativity of operators.*

- ADD_COL$_{p_1 \to p_2}$ [REM_COL$_{p_3}$ (*Table*)] $\equiv$ REM_COL$_{p_3}$ [ADD_COL$_{p_1 \to p_2}$ (*Table*)], *provided the sets of domains referred to in the parameter lists* p_1 *and* p_3 *are disjoint, and those in* p_2 *and* p_3.
- *Let* RES-OP *be either of* REM_COL *or* ADD_COL *and* NONRES-OP *be any non-restructuring operator, then* NONRES-OP$_{p_1}$ [RES-OP$_{p_2}$ (*Table*)] $\equiv$ RES-OP$_{p_2}$ [NONRES-OP$_{p_1}$ (*Table*)], *provided the sets of domains referred to in the parameter lists* p_1 *and* p_2 *are disjoint.*

Translation from *n*D-SQL to RRA: As stated earlier, the processing of *n*D-SQL queries is based on the translation of said queries into equivalent RRA expressions. For lack of space, we point the reader interested in details of the translation algorithm to [GL98]. We will provide here a very high level description of the algorithm.

Intuitively, we expect that the classical SQL parts of a 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. Restructurings are derived from both (i) the new FOR subclauses of the SELECT clause, and (ii) those conditions of the WHERE clause that involve some criteria.

The tables to which these operations will be applied are obtained from the instantiations of the variables (both those declared or those implicit that appear only after unfolding an abbreviation). We note that while the information necessary to instantiate non-tuple variables is contained in the catalog tables, we need to pull data by querying remote sources to instantiate tuple variables. We denote the Variable Instantiation Table containing the instantiations of a variable V_i by VIT_V_i.

Consider query Q8:

```
      SELECT T.Date, C.Measure,
             T.C AS R.Ticker FOR R.Ticker
(Q8)  FROM   mse -> R, mse::R T, mse::R -> C
      WHERE  R HASA Ticker AND C HASA Measure
             AND T.Date > 10|29|97
             AND R.Ticker > 'm' AND R.Ticker < 'n'
```

The following would be the equivalent RRA expression:

$\text{ADD\_COL}_{\text{VIT\_R.Ticker}\rightarrow\text{VIT\_T.Price}}$ {
REM_COL$_{\text{VIT_T.Measure}}$[
$\sigma_{\text{VIT\_T.Date}>10|29|97 \wedge \text{VIT\_R.Ticker}>'m' \wedge \text{VIT\_R.Ticker}<'n'}$(
VIT_R ⋈ VIT_T)] }

Note that it joins the necessary VITs, extracts the column criteria **Measure**, adds the criteria **Ticker** to the **Price** columns and applies the proper selections.

4.2 Optimization

Many opportunities for optimization arise from our use of RRA in processing *n*D-SQL queries. Since RRA is downward compatible with RA, and since the projections and selections are commutative with the new restructuring operators, we have the opportunity to push to remote sources some computations.

A preliminary step in optimizing the computations consists in ordering the instantiation of variables and using the technique of *sideways information passing* (sip). 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, we should delay instantiating the second variable until we can pass as bindings the values of the join attribute(s) obtained from the first variable's instantiations. These bindings would be passed on as selections in an SQL query.

We can also use for optimization purposes the following equivalences arising from the symmetry between our restructuring operators REM_COL and ADD_COL:

Theorem 4.2 *RRA expression equivalences.*

- $\text{ADD\_COL}_{p_1\rightarrow p_2}$ [REM_COL_{p_3} (*Table*)] $\equiv$ $\text{ADD\_COL}_{p_4\rightarrow p_2}$ (*Table*), *if* $p_3 \subset p_1 \wedge p_4 = p_1 - p_3$.
- $\text{ADD\_COL}_{p_1\rightarrow p_2}$ [REM_COL_{p_3} (*Table*)] $\equiv$ REM_COL_{p_4} (*Table*), *if* $p_1 \subset p_3 \wedge p_4 = p_3 - p_1$.
- $\text{ADD\_COL}_{p_1\rightarrow p_2}$ [REM_COL_{p_3} (*Table*)] $\equiv$ *Table*, *if* $p_1 = p_3$.

Another set of optimization rules rely on the following heuristic:

Heuristic 4.1 *It is in general more efficient to perform join or restructuring on fewer tuples, albeit they be wider.*

Since ADD_COL (in general) lowers the number of tuples and REM_COL increases it, we derive the following additional heuristics:

Derived Heuristics 4.1

- REM_COL_{p_3} [$\text{ADD\_COL}_{p_1\rightarrow p_2}$ (*Table*)] *is more efficient than* $\text{ADD\_COL}_{p_1\rightarrow p_2}$ [REM_COL_{p_3} (*Table*)].
- *If* $\bowtie_{p_1}$ *and* REM_COL_{p_2} *can commute and* p_2 *only refers to* $Table_2$, *then* REM_COL_{p_2} [$Table_1 \bowtie_{p_1} Table_2$] *is more efficient than* $Table_1 \bowtie_{p_1}$ [REM_COL_{p_2} ($Table_2$)], *provided the join selectivity is high.*[3]
- *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.*
- *If* $\text{AGG}_{p_1,\,p_2}$ *and* REM_COL_{p_3} *are such that* p_1 *and* p_3 *are disjoints but* $p_3 \subset p_2$ *(p_2 is the group-by list) then* REM_COL_{p_3} [$\text{AGG}_{p_1,\,p_2}$ (*Table*)] *is more efficient than* $\text{AGG}_{p_1,\,p_2}$ [REM_COL_{p_3} (*Table*)]

Another form of optimization 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 cascade, similar to the way join is a more efficient implementation of Cartesian product 'interleaved' with selection. In RRA, we have pinpointed two such series of operations: (1) A selection applied to the values of a column criterion without any restructuring being called for should be implemented more efficiently than by first removing the criteria, selecting on it, and adding it back. (2) A selection applied to the values of the concept of a complex column without any restructuring being called for should also be implemented more efficiently than by removing all criteria of that complex column, selecting on the concept and adding all criteria back. We define two new operators, Π^* and σ^* that capture the series of operations (1) and (2) respectively. For lack of space, we refer their formal definition to [GL98].

4.3 Processing of queries involving 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 recognizing that they involve the computation of 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 3.4 for another intersting example of a partial cube. The papers [Agar+96, ZDN97] discuss how algorithms for computing the CUBE can be adapted for computing partial cubes. Optimization of partial cubes is a topic of its own interest and is orthogonal to this paper. We mainly observe that queries with dimension variables and aggregation may in general involve: (i) computing a partial cube, and (ii) computing multiple visualizations of the result. The processing of such queries can be organized 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 is: how to interleave the computation of the partial cube with the required restructuring for each group-by in the partial cube.

5 Implementation

The implementation of our *n*D-SQL Server follows the architecture described in Section 4 (see Figure 6). The platform

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

is IBM PCs running Windows 95. The system is built as an external module, independent of the databases in the federation. The main components of the Server are: a Query Interface, a Translator to go from *n*D-SQL to RRA, an RRA Expression Optimizer, and an RRA Expression Executor. The Query Interface accepts a user's input query and verifies well-typedness, giving back helpful messages to the user if the query is ill-typed. Once a query is accepted by the Interface, the Translator creates the equivalent RRA expression which is sent to the Optimizer for a first pass. Then, the SQL queries for tuple_var VIT creation are created from the RRA expression and submitted to local databases, using sip to determine the order of submission, and passing parameters from one result to another query. When all the VITs are instantiated the Optimizer finishes optimizing the RRA expression. The Executor then executes it, using restructuring operations, and presents the final result to the user. The RRA operators are implemented in Visual C++.

6 Comparison With Related Work

We compare our work with previously proposed extensions to SQL, including SchemaSQL, and related work on multi-database query optimization.

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]), some motivated by querying OODBs ([KKS92, ASD+91, CL93]). An extensive comparison between *n*D-SQL and many of these languages appears in [GL98]. For lack of space, we merely observe that none of the above languages have both the restructuring and complex aggregation capabilities of *n*D-SQL. 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, and 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, OSQL, and XSQL are downward compatible with SQL. 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 *n*D-SQL; programming such features would thus be too low level and tedious. Some of the expressions for extracting domain values and values of criteria in *n*D-SQL resemble the path expressions of OQL [Cat96]. However, there 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 *n*D-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. This is far too limited compared to the restructuring capabilities of *n*D-SQL.

Finally, Ross [Ros92] and Gyssens et al. [GLS96] are two recently proposed algebras which have the power of manipulating meta-data. Of these, [Ros92] has limited restructuring capabilities, while [GLS96] has been shown to be complete for all generic restructuring transformations. However, both languages do not handle aggregation. A comparison between *n*D-SQL and a whole class of related logics is given in [GL98]. Ross et al. [SRC97] generalize CUBE into a multi-feature CUBE, and propose fast algorithms for computing queries involving this operator. Their contributions and those of this paper are complementary.

2. SchemaSQL: SchemaSQL is a multi-database interoperable query language proposed by one of the authors [LSS96], capable of restructuring and complex aggregations, and is the closest language to *n*D-SQL. In particular, our 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, this is a very serious problem. (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. Besides, unlike *n*D-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 *n*D-SQL, the notions of concepts and criteria are rich enough to always retain meta-data. (4) *Limited subaggregation*: SchemaSQL does not allow many subaggregates; 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 *n*D-SQL(e.g., see query (Q3), page 6). (5) *Multiple granularity*: One of the strengths of *n*D-SQL is its ability to express multiple granularity aggregation, possibly together with multiple visualizations (see Section 3.2), something SchemaSQL cannot do. On the query processing side, unlike [LSS96], we give an algebra and exploit its properties for query optimization.

3. Multi-database Query Optimization: Much work has been done in the context of multi-database query optimization, 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 optimization in multi-database systems. Our concern in query optimization in this paper is different: we focus on algebraic optimization of queries across multiple relational databases with heterogeneous schemas, where queries can involve attribute/value conflicts, restructuring, and complex OLAP-style aggrega-

tion. To our knowledge, optimization in such a setting is new. There are many interesting open research problems in this context, which we are currently investigating.

Acknowledgments

This research was supported by a grant from the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

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

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

[CD97] Surajit Chaudhuri and Umesh Dayal. 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.

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

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

[Flo95] Florescu, D. Using Heterogeneous Equivalences for Query Rewriting in Multi-Database Systems. In *Proc. 23rd Int. Conf. on Cooperative Information Systems*, 1995.

[GL97] Marc Gyssens and Laks V.S. Lakshmanan. A Foundation for Mulit-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. Design and Implementation of nD-SQL, a Multi-Dimensional Language for Interoperability and OLAP. Technical report, Concordia University, Montreal, Canada, in preparation.

[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 Joan S. Peckham, editor, *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.

[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.

[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] Xiaolei Quian. 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.

[SRC97] Kenneth A. Ross and Divesh Srivastava and Damianos Chatziantoniou. Complex Aggregation at Multiple Granularities. In *Proc. International Conference on Extending Database Technology (EDBT)*, March 1998.

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

[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.

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