Towards Framework for The Virtual Data Warehouse

Course-based MS Report by Yuan Ji

April 20, 2001

Table of Content

Abstract

This report presents a survey of data warehouse technology and an introduction of a framework for the implementation of virtual data warehouse. In the first par of survey, we briefly give a background of data warehouse and related concepts, and in the second part, we investigate query model and query language of data warehouse. Then we presented our virtual data warehouse with system design and implementation. One contribution of our work is proposed a new data warehouse query language XMDQL (XML Multidimensional Data Query language) that is in XML format.

1. Introduction and Data Warehouse Background

This essay is based on DIVE-ON (Datamining in an Immersed Virtual Environment Over Network) project, which is a system that utilizes virtual reality, databases, distributed computing to visualize data mining. The complete system consists of three parts: Virtual Data Warehouse(VDW), Visualization Control Unit (VCU) and User Interface manager (UIM). VCU and UIM provide data cube handling and interactive virtual mining tools to the user. And our work is mostly in VDW part, which is responsible for data warehouse management and query execution.

In this section, we will explain some basic concepts in data warehouse area, and data warehouse architecture plus multidimensional data model.

1.1 Data Warehouse

The data warehousing market is growing tremendously. According to a Surey.com report, “the worldwide spending was expected to rise from $37.4 billion 1999 to $148.5 billion by 2003. Also by 2003, the average amount of data that can be used for warehousing is expected to increase to 1.1 TB”[Hammond99]. Data warehouse technologies have been successfully deployed in many industries: manufacturing (for order shipment and customer support), retail (for user profiling an inventory management), financial services (for claims analysis, risk analysis, credit card analysis, and fraud detection), transportation (for fleet management), telecommunications (for call analysis), and healthcare (for outcomes analysis)[Chaudhuri97].

What is a data warehouse?

Ralph Kimball in his book “The Data Warehouse Toolkit”[Kimball96] states that a data warehouse is “a copy of transaction data specifically structured for query and analysis”. Data warehouse is informational and analysis and decision support oriented, not operational or transaction processing oriented.

William Inmon, who coined the term “data warehouse” in 1990, defined a data warehouse as a “subject oriented, integrated, time-variant, non-volatile collection of data that is used primarily in organizational decision making.”[Inmon92]

- **Subject oriented**. Subject-oriented in decision-support system is compared to application oriented in operational systems. In operational systems, data is organized to support specific business process, thus, the same data might be organized differently in different system. For examples, customer may be presented by different tables in traditional databases, but in data warehouse, we only have one customer object across the whole system. In other words, captures the basic nature of the business environment.

- **Integrated**. Integration is the most important aspect of the data warehouse environment, and it means data found within the data warehouse is integrated, or has been cleaned. It may appear in consistent naming conventions, in consistent measurement of variables, in consistent encoding structures, in consistent physical attributes of data, and so forth.

- **Time-variant**. Time variant has two issues. First, informational data as a time dimension, and it is different in operational environment as in data warehouse. The informational data is accurate when you access it, but data warehouse contains history data for last 5 to 10 years in most common siuations.

  Another point is that data warehouse data represents data over a long time horizon – from five to ten years, while the time horizon represented for the operational environment is much shorter - from the current values of today up to sixty to ninety days. That's because operational applications must do lots of transaction processes, so for performance they have to carry as few amount of data as possible. Therefore operational application data has a short time horizon as designed.

- **Nonvolatile**. We know the typical operational system only keeps data for a short period of time, e.g., 3 to 6 months, as it is only interesting for the daily business during this time span. And data is changing all the time. However, in a data analysis situation, data is kept or a pretty longer period, and after the data is in the data warehouse, there are no modifications to be made to this information, or very rare midiications.

Operational data vs. informational data

To compare data warehouse and operational databases, we also need to distinguish two kinds of data, operational data and informational data. Operational data is the data you use to run your business, and is typically stored in relational databases, but may be stored in legacy hierarchical or flat file formats as well.

Informational data is the data stored in data warehouse, and it's typically in a format that makes analysis much easier. Analysis can be in the form of decision support, report generation, executive information systems, and more in-depth statistical analysis. Informational data comes from operational data, after some preprocessing, like data cleaning, integrating.

OLTP vs. OLAP

Not only data are different in data warehouses and in operational systems, but also the tools. The data warehouse supports on-line analytical processing (OLAP), which requires quite different functions from the on-line transactional processing (OLTP) applications.

OLTP applications typically automate structured and repeated data processing tasks for day-to-day operations of companies such as sales transactions, banking transactions. These tasks consist of short, atomic, isolated transactions with detailed, up-to-date data and often by reading or writing a few records in their relational databases. Consistency and recoverability of the database are critical, and maximizing transaction throughput is the key performance issue.

Data warehouses, in contrast, are targeted for decision support, and are maintained separately from the companies' operational databases. OLAP tools only handle historical, summarized and consolidated rather than detailed, individual records.

Advantage of Data Warehouse

[Sakaguchi96] surveyed 456 articles about data warehouse and gave the following advantage of data warehouse from those articles:

- **Simplicity**. This is the highest frequently mentioned advantage. Because of its subject-oriented feature, data warehouses provide a single image of business object by integrating various operational data sources. So it makes existing legacy systems still useful and combines inconsistent data from many legacy systems into one coherent set.

- **Quality Data**. Data warehouse gives better quality data such as consistency, accuracy, and documentation.

- **Fast Access**. Since data warehouses integrate all data from different data sources in one place, users don't need to login many systems, and response time should be reduced.

- **Easy to Use**. Because data warehouses copy some of the operational data and put in a separate database, queries from users do not interfere with normal operations, and with the help of OLAP tools, it's much easier for decision makers to access business data.

- **Separate decision-support operation from production operation.** As we said before, data warehouses separate operational, continually updated transaction data from historical, more static data required for business analysis. So managers and analysts can use historical data in data warehouse for their decision-making activities without slowing doing down the production operation in operational systems.

- **Gives competitive advantage**. Data warehouses help business entities become more competitive, better understand customers, and more rapidly meet market demands by providing better-organized information.

- **Ultimate distributed database.** Data warehouses gather information from disparate and potentially incompatible locations throughout the company and put it to good use. And we can use middle-ware or other client/server tools to link hose disparate data sources to an ultimate distributed database.

- **Operation cost.** After building a real data warehouse, it becomes easier to integrate new operational systems, and information-technology group in the company will generally require fewer resources.

Disadvantage of data warehouse

- **Complexity and anticipation in development.** You cannot just buy a data warehouse; you have to build one because each warehouse has a unique architecture and a set of requirements from the individual needs of the organization. That means you cannot apply existing data warehouse to new situation; you have to create from scratch.

- **Takes time to build.** To build a data warehouse takes time. Maybe after you finished, the data warehouse was out of date.

- **Expensive to build. D**ata must be moved or copied from existing databases, sometimes manually, and data needs to be preprocessed into a common format. Data preprocessing may includes data cleaning, integration, transformation, reduction or discretization. And these tasks are very expensive.

- **Lack of API.** Data warehousing software still lack a set of application programming interfaces (API) like ODBC in relational databases.

    - **End-user training.** Users of data warehouse require training to capitalize on those data analysis provided by data warehouse.

1.2 Data Warehouse Architectures

The data warehouse can be architected in a variety of ways:

Data mart

Data marts, or localized data warehouses, are small sized data warehouses, and typically created by individual departments or divisions to provide their own decision support activities. For example, a data mart can be created for specific products or functions, like electronics or customer management. In another example, data mart may be created for user populations with the same technical environments like Windows system, Unix system, etc.

These data marts come from operational databases in departments or regions of companies, and may also pull some data from other departments or divisions. After finished building them, these data marts are no longer coordinating, because different data mart has different structure, and user requirements focus in different areas. Maybe some of them can communicate with each other for consolidation and global reporting, but they are independent data warehouses.

The purpose to build a data mart is to get prototype as soon as possible without waiting for a larger corporate data warehouse, because it’s small and easy to develop. Stand-alone data marts can be used by organizations with very independent and "nonintersecting" departments as a starting point in an overall strategy for a centralized corporate data warehouse. But after having several data marts, organizations couldn’t use them corporately, because they are not consistent with each other. So data marts are only used before building real data warehouse for fast prototype and evaluation.

Central data warehouse

The centralized data warehouse, popularized by IBM's Information Warehouse, is what we called data warehouse in common meaning. This central data warehouse copies and stores all operational and external data and adheres to a single, consistent enterprise data model. This central data warehouse may be generated from many individual data marts for better performance and easy accessibility.

Since central data warehouse only has one data model, the data are consistent and complete to users in this architecture. So users only have one environment to logon, without worrying about data in different platforms and environments. And most of the processing is done at the corporate site, therefore it’s very useful or enterprises.

The main drawback of the central data warehouse is that, as we talked before, it is quite difficult to develop a global data model for most organizations, because finding a single structure for everything is tough. It is also difficult to agree on a corporate wide level of detail and naming conventions. Organizations must also carefully manage the performance and end-user access to centralized warehouses to ensure that the users continue to rely on the centralized data warehouse

Distributed data warehouse

Like distributed file systems and distributed database systems, data warehouses can be distributed in an enterprise. These distributed data warehouses (distributed data marts) are consist of many local data warehouses (data marts) and can be accessed by the end users through a "warehouse front end". This front end is a kind of software with a global data image. It also knows the location and format of the needed data and how to send the queries to the final destination. When user asks for something, this software will find the local data warehouse to perform the query and get the result back.

With the benefits of distribution, distributed data warehouses may be more flexible, have higher performance and load balancing. Besides, it can be designed to match the topology of many organizations, such as a corporate data warehouse at the corporate site and regional data warehouses (the data marts) at the branch offices. So corporate queries will go to corporate ones, and local queries will go to regional ones. And different technologies can e applied to different data warehouses to better match their subjects.

Distributed data warehouse still need a global data model, which is, like central data warehouse, hard to develop. Besides, global dictionary and topological information make it even more complicated. Another disadvantage is performance. If the data warehouses are widely distributed, significant performance degradation and service outages can occur. So it requires considerable trade-offs in data distribution and query optimization."

Middle-ware approach: Virtual Data Warehouse

All above three architectures need creating real data warehouse systems, that is, you have to get system requirements, analyze existing database or legacy systems, develop data warehouse model for the local divisions or the whole enterprise. This is a very time consuming task and existing systems are outside of the new data warehouse. How to build a data warehouse connected with run-time database or legacy systems becomes an interesting topic in the early 1990s, and middleware approach provides an alternative to traditional data warehousing, called virtual data warehouse.

Generally, virtual data warehouses use middleware to build direct connections among disparate applications, whereas traditional data warehouse provides a central repository for enterprise data. Middleware is software acting as data hubs and allowing access to the corporate data stored in heterogeneous data sources, like relational databases, legacy systems.

The most popular approach of virtual data warehouse is "wrapper" or "surround' data warehouse, that is the old legacy operational data sources are "wrapped" or "surrounded" by middleware where there are no changes made to the underlying operational systems. Ideally, no preprocessing is required to old system; no programs need be coded to transform legacy data; no integrating and cleaning. Instead, you just buy some software packages that can "wrap" your legacy systems to read data from them, very cheap and fast. And like distributed data warehouse, virtual data warehouse still relies on the creation of an independent meta-data definition of the corporate data, or global data model, and therefore, as the same easy-of-use advantages without the complexity of building a traditional data warehouse system.

However, this approach was criticized by Inmon in the "Virtual Data Warehouse: The Snake Oil Of The Nineties"[Inmon96]. "…[Snake oil] was pleasing at the moment it was consumed. But it had no real curative effects, and ultimately was a disappointment to those who bought it in the hopes that it would do something real."[Inmon96]

Inmon indicated virtual data warehouse approach failed in several fundamental ways:

- **Performance**. Operational systems are running for transaction tasks, but OLAP tools may ask for 10,000,000 rows of data in the middle of the on-line transaction processing, and no good database designer wants to design databases for that.
- **Historical data**. Time-variant of data warehouse means a rich amount of historical data, typically 5 to 10 years worth, which providesa foundation for analysis and slicing_and_dicing data. But operational systems under virtual data warehouse only contain a minimal amount of history – 30 to 60 days typically. As such it cannot organize and manage the historical data because no historical data there.
- **Data structure**. The structures in the operational systems are designed to suit the needs of operational processing, without any idea of data analysis, and the "wrapper" software around them can do nothing for optimizing the structure of the data for OLAP.
- **Integration**. The middleware generally will give you access only to data in its raw form. There is no integration in the legacy systems beneath the virtual data warehouse, so OLAP tools that accesses the old legacy data must do the integration by their own.
- **Aggregation**. Data warehouse contains summary data, while operational system does not.

Those significant disadvantages impeded the development of virtual data warehouse.

Hybrid Virtual Data Warehouse

However, virtual data warehouse techniques are still attracting to us. So we considered mixing distributed data warehouse and pure virtual data warehouse to hybrid virtual data warehouse. According to the name, hybrid virtual data warehouse also works like middleware, but the data sources may be data marts or legacy systems. In our approach, historical data is stored in local data warehouses or corporate data warehouses, and we can also connect to operational system for fresh data.

Since most data comes from data marts, only small part of it is pulled from legacy system, the performance is not a big issue. We may still have the problem of data structure, integration, thus the middleware has to be designed carefully, and we can use the functionality in the OLAP model to do data cleansing.

1.3 Multidimensional Data Model and OLAP operations

Measures and dimensions

The most popular conceptual model for OLAP is the multidimensional view of data in the warehouse[Chaudhuri97]. For example, we want to build a data warehouse for a grocery store company, the everyday transaction records contain which customer bought which product at what time. The goods price and soled item amounts are called numeric *measures* that are the objects of analysis in a multidimensional data model. The categorical attributes like product, date, and customer are called *dimensions*. Each of the numeric measures depends on a set of dimensions and the dimensions together are assumed to uniquely determine the measure. Thus the multidimensional data views a measure as a value in the multidimensional space of dimensions.

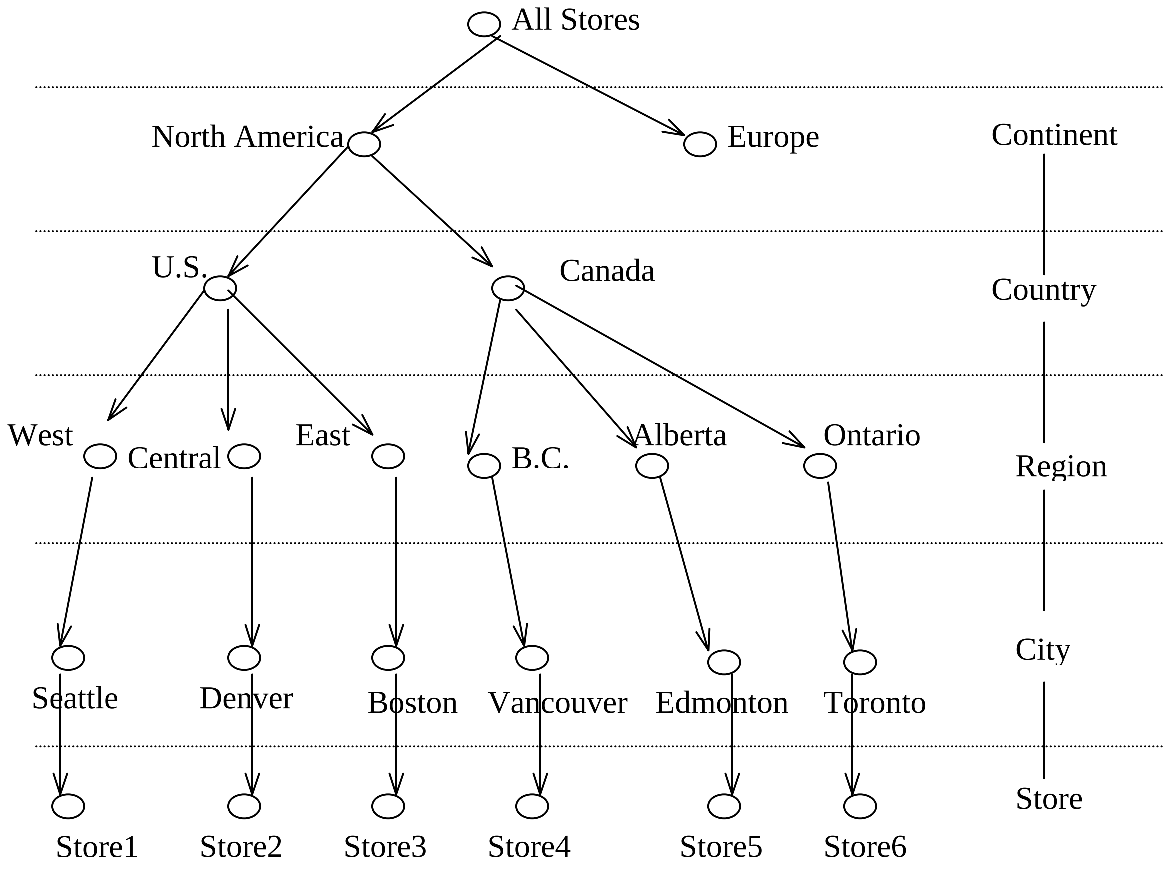


Figure 1.1 Store Hierarchy

Dimensions usually have associated with hierarchies, for example, the Store dimension may consist of five aggregation levels: store name, city, region, country, and continent. And the attributes or levels of this dimension may be related via a hierarchy of relationship, as above figure 1.1 showed.

Time is a special dimension that is of particular significance to decision support, because time dimension has partial order. So it is often treated differently. As in figure 1.2, Time dimension have Year, Quarter, Month, Week and Day levels. But the year can be divided into weeks, then days, or quarters, months then days. So we may have multiple routes from one level to another.

$$\begin{tikzcd}
\text{Year} \arrow[r] \arrow[drr] & \text{Quarter} \arrow[r] & \text{Month} \arrow[r] & \text{Day} \\
& & \text{Week} \arrow[ur] &
\end{tikzcd}$$

Figure1.2 Time Dimension Partial Order

Multidimensional data cube

Now we have the concept of data cube. The OLAP data cube can be conceptually viewed as a multi-dimensioned cube representing any number of descriptive categories (dimensions) and quantitative values (measures) as following figure 1.3. The OLAP data cube contains summaries of selected detail information from your existing database and saves it as a multi-dimensional array, rather than the common 2-dimension relational tables.

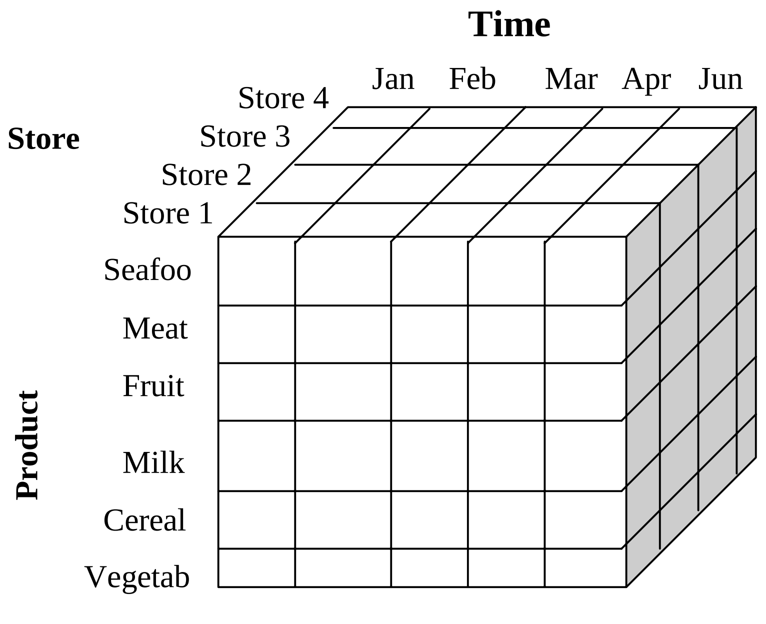


Figure 1.3 Multidimensional data cube

OLAP operations

Having multidimensional data cube, an analyst might want to see a subset of it with some attributes and restricted values. Or he/she might need other interactive data querying. These operations, which are part of decision support, are called On-Line Analytical Processing (OLAP).

To rotate the data cube to show a particular face is called *pivoting*. This operation is often supported by the multidimensional spreadsheet applications. For example, we may select tow dimensions Store (in tore level) and Time (in year level) for pivoting, and aggregate sales measure. So the aggregated values are displayed in the two dimensional spreadsheet, with store names as row headers and years as column headers. The value in the grid (x,y) coordinate corresponds to the aggregated sales value of store x and year y.

Other operators related to pivoting are *rollup* or *drill-down.* Rollup corresponds to aggregating current data on one of the dimensions, for example, aggregating the sales data in Store dimension from city to region. The drill-down operation is the converse of rollup. Thus, drilling-down the Store dimension from region to city gets the sales value for each city, and further drill-down will get the sales or each individual stores.

Slice_and_dice corresponds to reducing the dimensionality of the data, i.e., selecting some subset of the cube. For example, if we slice_and_dice data cube in figure1.3 for a specific product, we can get a table with dimensions store and month.

Database design

Database designers in OLTP environment often use Entity Relationship (ER) diagrams and normalization techniques. However, the database schemas created by ER diagram are not suitable for decision support systems, where querying and loading aggregated data need more efficient structure. Most data warehouses use a *star schema* to represent the multidimensional data model. The database consists of a single fact table and a single table for each dimension. Each tuple in the fact table consists of a pointer (foreign key) to each of the dimensions that provide its multidimensional coordinates, and stores the numeric measures for those coordinates. Each dimension table consists of columns that correspond to attributes of the dimension. Figure 1.4 shows an example of a star schema.

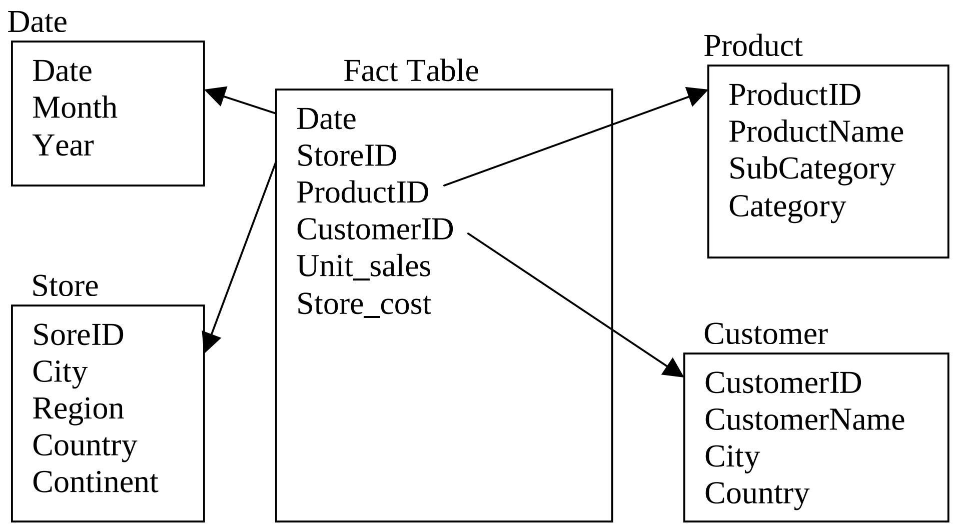


Figure 1.4 Star Schema

Implementation Architectures

There are two main approaches used to build multidimensional databases. One approach maintains the data as k-dimensional matrix based on a non-relational specialized storage structure, and stores into Multidimensional OLAP (MOLAP) servers. And while building the MOLAP, the database designer also compute all useful aggregations for roll-ups, so roll-ups and drill-downs are answered in the interactive time.

Another approach implements data warehouse on a relational backend, called Relational OLAP (ROLAP) servers. These ROLAP servers support extensions to SQL and use indexes built on materialized views to efficiently do operations.

2. Data Warehouse Query Language

To end users, the data in data warehouse is in multidimensional status, no matter physically stored in ROLAP or MOLAP servers. On one hand, decision makers want to do OLAP operations by graphical user interface tools. On the other hand, sophisticated users may need a declarative, igh-level query language to perform complex queries. Many extended SQL languages or multidimensional query languages are proposed in recent years. Here we will introduce Cube operator and two other query models.

2.1 SQL and Data Cube Operator

Gray et al. [Gray95] first proposed an extension to SQL with a *Cube* operator to support multidimensional query.

SQL aggregation function

Data warehouse usually refers to huge amounts of data, and data analysis applications look for usual patterns in those data, so they can extract relevant data from the warehouse, aggregate it and analyze the results.

Data extraction and aggregation are common in SQL statements. The SQL standard provides five functions to aggregate the values in a table: `COUNT(), SUN(), MIN(), MAX()`, and `AVG()`. In addition, SQL allows aggregation over distinct values by `DISTINCT`. In many SQL systems, even more functions are provided, such as statistical functions (median, standard deviation, variance, etc.), physical functions (center of mass, angular momentum, etc.), financial analysis (volatility, Alpha, Beta, etc.), or even user-defined functions.

Problems with GROUP BY

The GROUP BY relational operator partitions a table into disjoin tuple sets and then aggregates over each sets as illustrated in following figure 2.1[Gray95]:

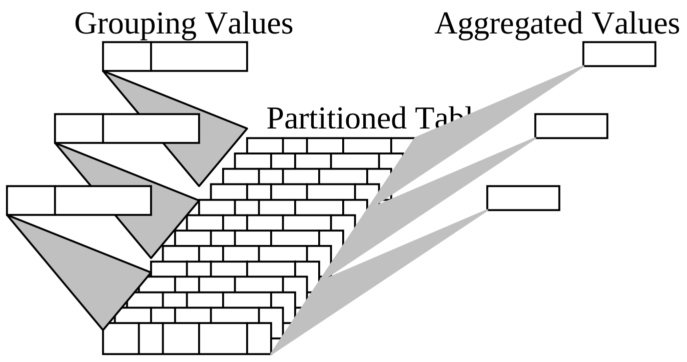


Figure 2.1 GROUP BY Operator

There are three common difficulties in data analysis:

1. **Histograms**. The standard SQL `GROUP BY` operator does not allow a direct construction of histograms (aggregation over computed categories).

2. **Roll-up Totals and Sub-Totals for Drill-downs**. You have to store each level i.e. subtotal of the aggregation.

3. **Cross Tabulations**. Building a cross-tabulation with SQL is even more daunting since the result is not a really relational object.

Data Cube operator

Jim Gray et al. [Gray95] proposed an extension to SQL with a new operator, Cube, to generalize the N-dimensional group-by function.

The data cube operator builds a table containing all the aggregate values. The total aggregate is represented as the tuple:

```
  ALL , ALL, ALL, …, ALL, f(*)
```

Where `f(*)` is an aggregation function.

The original SQL `GROUP BY` syntax is:

```
GROUP BY
{<column name> [collate clause] , …}
```

And the extended SQL GROUP BY operator becomes:

```
GROUP BY
      { (<column name> | <expression> )
      [ AS <correlation name> ]
      [ <collate clause]
      , …}
      [ WITH (CUBE | ROLLUP)]
      )
```

For example, we have a relational table of SALES of cars (example comes from [Gray95])

Table 1 SALES

Model	Year	Color	Sales
Chevy	1990	red	5
Chevy	1990	white	87
Chevy	1990	blue	62
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	blue	49
Chevy	1992	red	31
Chevy	1992	white	54

Chevy	1992	blue	71
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	blue	63
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	blue	55
Ford	1992	red	27
Ford	1992	white	62
Ford	1992	blue	39

And the SQL statement is:

```
SELECT Model, Year, Color, SUM(sales) AS Sales
FROM Sales
WHERE Model in {‘Ford’, ‘Chevy’}
      AND  Year BETWEEN 1990 AND 1992
GROUP BY Model, Year, Color WITH CUBE
```

The result DATA CUBE table is like this:

Table 2 Data Cube

Model	Year	Color	Sales
Chevy	1990	blue	62
Chevy	1990	red	5
Chevy	1990	white	95
Chevy	1990	ALL	1554
Chevy	1991	blue	49
Chevy	1991	red	54
Chevy	1991	white	95
Chevy	1991	ALL	198
Chevy	1992	blue	71
Chevy	1992	red	31
Chevy	1992	white	54
Chevy	1992	ALL	156
Chevy	ALL	blue	182
Chevy	ALL	red	90
Chevy	ALL	white	236
Chevy	ALL	ALL	508
Ford	1990	blue	63
Ford	1990	red	64
Ford	1990	white	62
Ford	1990	ALL	189
Ford	1991	blue	55
Ford	1991	red	52
Ford	1991	white	9
Ford	1991	ALL	116
Ford	1992	blue	39
Ford	1992	red	27

Ford	1992	white	62
Ford	1992	ALL	128
Ford	ALL	blue	157
Ford	ALL	red	143
Ford	ALL	white	133
Ford	ALL	ALL	433
ALL	1990	blue	125
ALL	1990	red	69
ALL	1990	white	149
ALL	1990	ALL	343
ALL	1991	blue	106
ALL	1991	red	104
ALL	1991	white	110
ALL	1991	ALL	314
ALL	1992	blue	110
ALL	1992	red	58
ALL	1992	white	116
ALL	1992	ALL	284
ALL	ALL	blue	339
ALL	ALL	red	233
ALL	ALL	white	369
ALL	ALL	ALL	941

`CUBE` operator first aggregates over all the `<select list>` attributes, such as Model, Year, Color here, as in a standard `GROUP BY`. The result is the last record of the above table. Then it substitutes `ALL` for each aggregation columns with attribute values – super aggregation of the global cube. If there are N attributes in the select list, there will be $2^N - 1$ super-aggregate values. And suppose the different value account of each attributes are $C_1, C_2, \ldots, C_N$ then the cardinality of the resulting cube relation is $\prod (C_i + 1)$. For above example, the table 1 has $2 \times 3 \times 3 = 18$ rows, while the derived data cube table 2 has $3 \times 4 \times 4 = 48$ rows. And the respective sets are:

```
Model.ALL = ALL(Model) = {Chevy, Ford}
Year.ALL  = ALL(Year)  = {1990, 1991, 1992}
Color.ALL = ALL(Color) = {red, white, blue}
```

The `ALL` value is a non-value, like `NULL`, and `ALL` becomes a new keyword denoting the set value.

If the application wants only a roll-up or drill-down report, the full cube is too huge to compute. It is reasonable to offer another function `ROLLUP` in addition to `CUBE`. `ROLLUP` produces just the super-aggregates:

```
(f1, f2, …, ALL),
      ...
(f1, ALL, …, ALL),
(ALL, ALL, …, ALL).
```

After defining the `CUBE` operator, they also discussed how to address the data cube and proposed some ways to compute the data cube.

2.2 Agrawal's Multidimensional Data Model

SQL extension with CUBE operator cannot solve all difficulties in multidimensional databases. So several new multidimensional data models and query languages are proposed. Rakesh Agrawal et al. proposed a data model and few algebraic operations for multidimensional databases [Agrawal97].

Data Model

Their data model is derived from multidimensional cube model, and has some outstanding features [Agrawal97]:

- It is a multidimensional cube with a set of basic operations designed to unify the divergent styles and to extend the current functionality.
- It treats all dimensions and measures symmetrically. The model also is very flexible in providing support for multiple hierarchies along each dimension and support for ad hoc aggregates.
- The input of each operator is defined on the cube and output of it is also a new cube. Thus the operators are closed and can be freely reordered.
- It keeps the operators number minimal. That means no operator can be expressed in terms of others nor can any one be dropped without sacrificing functionality.
- The modeling framework separates the front-end graphical user interface used by a business analyst from the backend storage system used by the corporation.

The most interesting part of the model is symmetric treatment to dimensions and measures. It makes the logical model nice to look at, and operations on the logical model only need to handle one type of data. However, it might make the cube much larger than the original one, and harder to implement. The model also provides support for multiple hierarchies along each dimension, but it's not clear in the paper.

In the logical model, data is organized in one or more multidimensional cubes. A cube has the following components:

- ***k* dimensions.** A name D_i and a domain dom_i from which values are taken for each dimension.
- **Elements.** It is a mapping $E(C)$ from $dom_i \times dom_2 \times \cdots \times dom_k$ to an n-tuple, 0 or 1. The element at 'position' $d_1,...,d_k$ of cube C is $E(C)(d_1,...,d_k)$. Therefore, the model does not require the dimensions to have a ranked, discrete domain.
- An **n-tuple** of names that describes the n-tuple element of the cube.

In this model, measures are also dimensions, so the data cube elements is different from traditional cube values, and cube has more logical dimensions than physical one. The elements of a cube can be 0, 1 or an n-tuple $< X_1,...,X_n >$. A 0 of the element corresponding to $E(C)(d_1,...,d_k)$ means that combination of dimension values does not exist in the database. A 1 indicates the existence of that particular combination. And an n-tuple indicates that additional information is available for it. 1 and n-tuple cannot be mixed in one cube, that is if any element is 1, none element canbe n-tuple, and vice-versa. Empty cube is that all the elements of a cube are 0.

Operators

The operators use a cube C with *k* dimensions as $D_i,...,D_k$. D_i also refers to the domain of dimension D_i or we use $dom_i(C)$ if it's not clear in the context. The lower case letters like a, b, c refer to constants.

First tow new function f_{elem} and f_{merge} are specified. f_{elem}, also called *element combining function*, combine several element values into one value. And f_{merge}, *dimension merging functions,* can merge values along a dimension.

Here are the operators defined in [Agrawal97]:

- **Push**. The push operation converts dimensions into elements that can then be manipulated using function f_{merge}. This operator is needed to allow dimensions and measures to be treated uniformly.
- **Pull**. The converse of the push operation is pull. Pull creates a new dimension for a specified member of the elements. The operator is useful to convert an element into a dimension so that the element can be used for merging or joining.
- **Destroy Dimension**. This operation eliminates a dimension D that only has one value in its domain. The presence of a single value implies that for the remaining *k* - 1 dimensions, there is a unique *k* – 1 dimensional cube. So removing it we get a unique *k* –1 dimensional cube.
- **Restriction**. This operator removes the cube values of operated dimension that does not satisfy a stated condition.
- **Join**. The join operator is used to relate information in two cubes. For example, joining an *m*-dimensional cube C1 with an *n*-dimensional cube C2 on *k* dimensions, called *joining dimensions*, will get a new cube C_a with *m*+*n*-*k* dimensions.
- **Merge**. It is an aggregation operation. Merging elements on one dimension can probably produce a smaller domain dimension, if multiple elements in the original cube are mapped to the same eleet in the new cube.

The operators proposed in [Agrawal97] have similarity with relational algebra by design. The authors wanted to explore how much of the functionality of current multidimensional products can be abstracted in terms of relational algebra. And by developing operators that can be translated into SQL, they hope to create a fast path for providing OLAP capability on top of relational database system.

2.3 Cabibbo's Multidimensional data model *MD* and Query Language

Another good multidimensional data model is *MD* proposed by Cabibbo, et al, [Cabibbo97] [Cabibbo98] [Cabibbo00]. They also introduced three query language *MD-A, MD-C*,, and *MD-G* based on this data model.

The *MD* data model

Their main contribution is *MD*, a logical model for OLAP databases "that, unlike other multidimensional data models, is independent of any specific implementation and as such provides a clear separation between practical and conceptual aspects."[Cabibbo00]

The Multidimensional data model (*MD* for short) is much different from other data models we talked before. *MD* doesn't use cube to represent multidimensional data, but f-table, which has more meanings than that in star-schema. Each f-table not only represents a traditional fact table, but also is a function, from coordinates to measures. So the 'f' in the term f-table stands for "function" and "fact".

Another main construct in *MD* is dimension, which is the same in cube models. Dimensions are the categories we used to analyze business data. Each dimension has a hierarchy of levels, with descriptions associated. And values of different levels are related by *roll-up* functions, so *MD* also support partial orders.

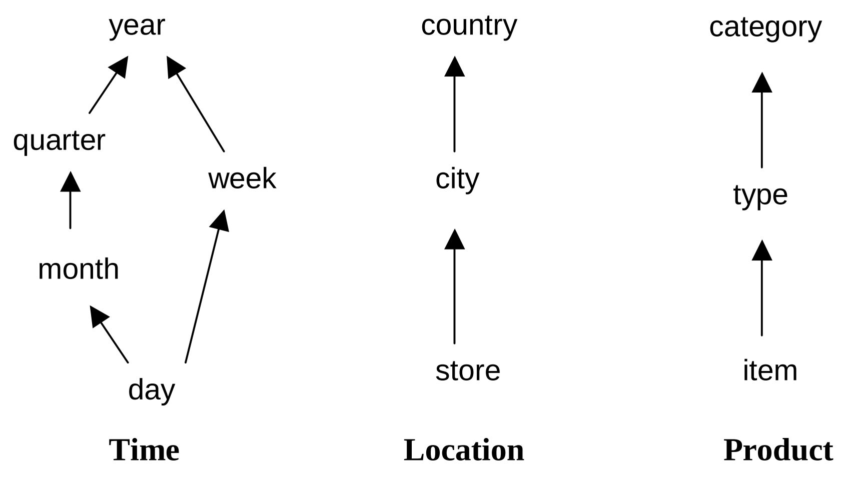
MD model consists of several concepts [Cabibbo00]:

- **Level**. An *MD* level *l* is a countable set of values. Different levels are associated with pair wise disjoint sets of values.
- **Dimension**. An *MD* dimension is a triple $(L, \leq, R)$ of a finite set of levels; a partial order on the levels and a family of roll-up functions.
- **Scheme**. An *MD* scheme is a triple (D, F, LD) of a finite set of dimensions; a finite set of f-table schemes and a finite set of level descriptions.
- **Coordinate and Instance**. Let $S=(D, F, LD)$ be an *MD* scheme and $f[A_1,:l_1, \ldots, A_n:l_n]:l_0$ be an f–table schema in *F*. A (symbolic) *coordinate* over an f-table scheme $f[A_1,:l_1, \ldots, A_n:l_n] \rightarrow \langle M_1:l_1', \ldots, M_m:l_m' \rangle$ in *F* is a function mapping each attribute name $A_i (1 \leq i \leq n)$ to an element in l_i. An instance over f is a partial function that maps coordinates over f to tuples over $<M1,:l1', \ldots, M_m:l_m'>$.

Example

We continue using a chain grocery store company as an example to explain the *MD* model. Suppose the grocery store business data has dimensions like **time**, **product** and **location**. The time dimension may be organized in levels day, month, quarter, year and week. For instance, April 20, 2000 is an element of the level day. Elements of level day roll-up to elements of level month, but also to elements of level week.

dimensions



f-table scheme

Sales[*Period*:day, *Product*: item, *location*: store] → ⟨*NSales*: numeric, *Income*: numeric⟩

CostOfItem[*Product* : item, *Month* : month]→⟨*Cost* : numeric⟩

Level Description

Address (store) : string

Figure 2.2 The Grocery Store Scheme Example

Above figure 2.2 shows the *MD* scheme, over dimensions time, location and product. This scheme has two f-tables, named Sales and CostOfItem, and one level description, *address*. The f-table Sales describes some summary data for the sales transactions, organized along dimensions **time** (at day level), **location** (at store level), and **product** (at item level). The measures for this f-table are *NSales* (the number of items sold) and *Income* (the gross income), both having type numeric. Because the costs of items are changing from month-to-month, we use the f-table CostOfItem to represent. The level description *address* is used to associate textual information to the elements of the level *store*.

A possible instance for this model scheme example is shown in figure 2.3.

SALES

Period	Product	Location	NSales	Income
April 1, 2000	milk	S Edmonton	3	12.98
April 1, 2000	egg	S Edmonton	10	51.34
April 1, 2000	bread	W Edmonton	3	4.67
April 1, 2000	banana	W Edmonton	23	4.54
May 1, 2000	milk	S Edmonton	3	14.21
May 1, 2000	egg	S Edmonton	6	20.11
May 1, 2000	milk	Calgary	3	14.21
May 15, 2000	bread	Calgary	2	2.56
May 15, 2000	egg	W Edmonton	5	25.53

COSTOFITEM

Cost	April-2000	May-2000	June-2000
milk	3.99	4.39	4.99
egg	1.99	2.29	2.59
bread	0.99	1.99	2.49
banana	0.39	0.49	0.59

Address

Store	
S Edmonton	Calgary Trail
W Edmonton	Stony Plain Rd
Calgary	23 Ave.

Figure 2.3. Instance of Grocery Store Scheme

Note that two different representations for f-tables are used in the figure. A symbolic coordinate over the f-table SALES is [day : April 1, 2000, item : milk, location : S Edmonton]. The actual instance associates with this entry the value 3 for the measure *NSales* and the value 12.98 for the measure *Income*. The figure also shows a tabular representation for the level descriptions address.

It is apparent that the notion of "symbolic coordinate" is related with that of "tuple" in the relational model. It can also be noted that the notation used for symbolic coordinates resembles subscripting into a multi-dimensional array (although in a non-position way).

Querying *MD* database

An *MD* query is a mapping from instances over an input *MD* scheme to instances over an output *MD* scheme. The input and output schemes are defined over the same dimensions but different f-tables.

[Cabibbo00] introduced three query language *MD-A*, *MD-C*,, and *MD-G*. Before we discuss these query languages, we assume there is an f-table EDMONTONSALES (the output f-table) with scheme:

EDMONTONSALES [*Period* : day, *Product* : item, *Location* : store] →⟨*NSales*: numeric⟩,

This output f-table represents the number of sales for each item in each day, only for the stores in Edmonton. This sales data can be calculated from the input f-table SALES, having scheme

SALES[*Period*: day, *Product*: item, *location*: store] → ⟨*NSales*: numeric, *Income*: numeric⟩

Algebraic query language *MD-A*

Like relational algebra query language, *MD-A* is an algebra based on a set of operations over f-tables, manipulating the –tables in procedural way. The operators include Cartesian Product, Natural Join, Roll-up, Level Description, Selection, Simple Projection, Aggregation, Abstraction.

Because we want the sales data for stores in Edmonton, we first extend the input f-table with a new attribute over the level city by roll-up operator $\rho_{A1:l1}^{A2:l2}(F)$. Then we perform a selection over the new attribute city, which equals "Edmonton", and finally project out the additional attribute and unneeded measures. The algebra expression is as following:

$$\pi_{[Period, Product, Location] \to \langle NSales \rangle}\left(\sigma_{City=Edmonton}\left(\rho_{Location:store}^{City:city}(SALES)\right)\right)$$

Calculus query language *MD-C*

The calculus query language *MD-C* is a first-order calculus for f-tables, and allows expressing queries in a declarative way.

An *MD-C* query whose output f-table has scheme $f[A_1:l_1, \ldots, A_n, l_n] \to \langle M_1:l_1', \ldots, M_m:l_m' \rangle$ can be specified by means of an expression of the following form:

$$\{x_1, \ldots, x_n: y_1, \ldots, y_m \mid \psi(x_1, \ldots, x_n, y_1, \ldots, y_m)\}$$

The query EDMONTONSALES can be specified by means of the following calculus expression:

$$\{\text{day, item, store: NSales} \mid \exists \text{income}(\texttt{SALES[day,item,store]=}\langle\texttt{NSales, Income}\rangle \wedge \rho_{store}^{city}(store) = Edmonton\ )\}$$

Graphical query language *MD-G*

The graphical query language *MD-G* provides an interactive way for end-users to query on a multidimensional database. It describes an *MD-G* f–table with a graph called f-graph. For example, the f-graph for SALES is shown in figure 2.4.

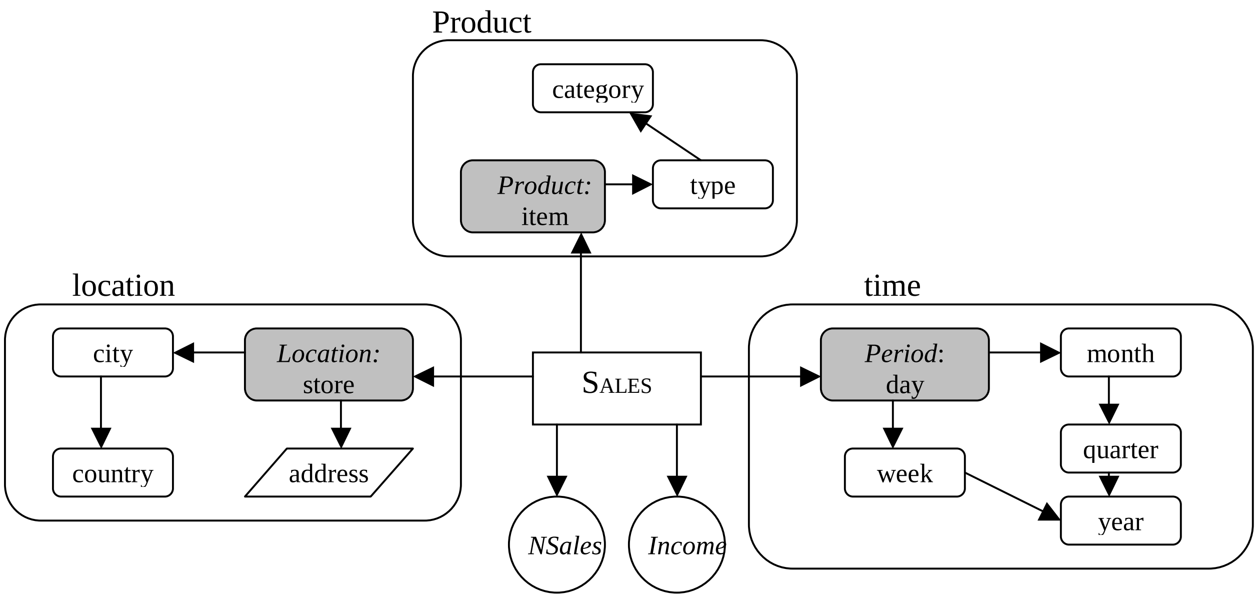
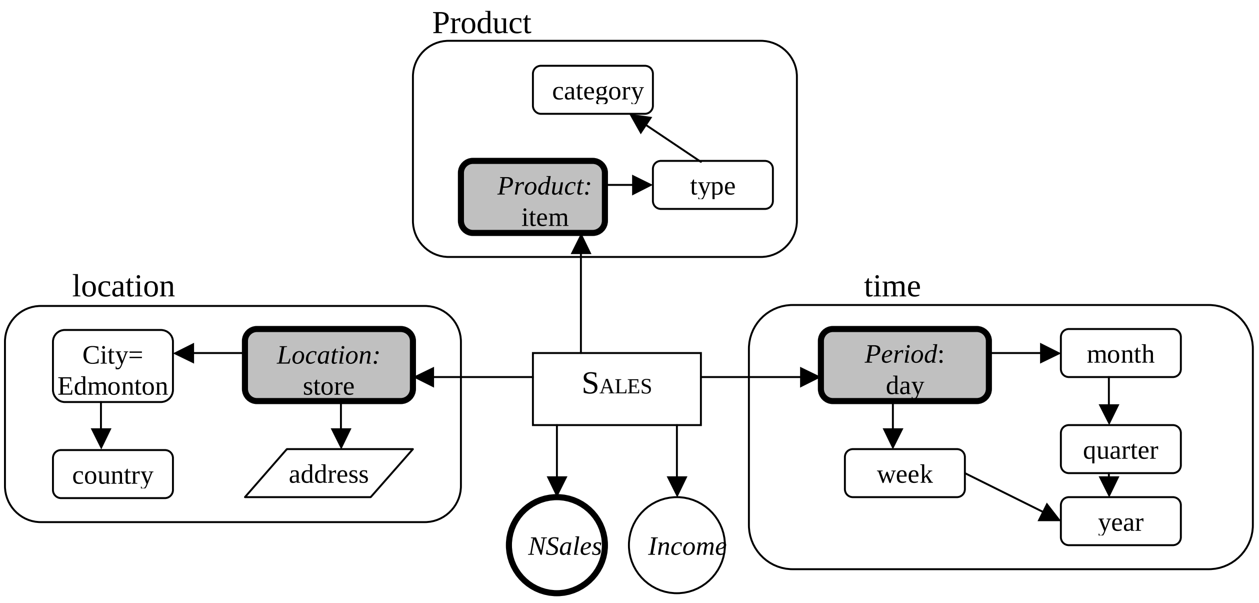


Figure 2.4 f-grpah for SALES

The central, rectangular node is *f-node* to represent the f–table. Ovals denote levels of dimensions and parallelograms denote level descriptions. An arc between two levels represents a roll-up function, and arc between an f-node and a measure node associates the f-table with measure.

In *MD-G*, the query is specified by means of several f-graphs to restructure schema. For EDMONTONSALES example, the input –graph is in figure 2.4, and the restructuring graph is shown on the bottom of figure 2.5, and the output f-graph is shown on the bottom of figure 2.5.



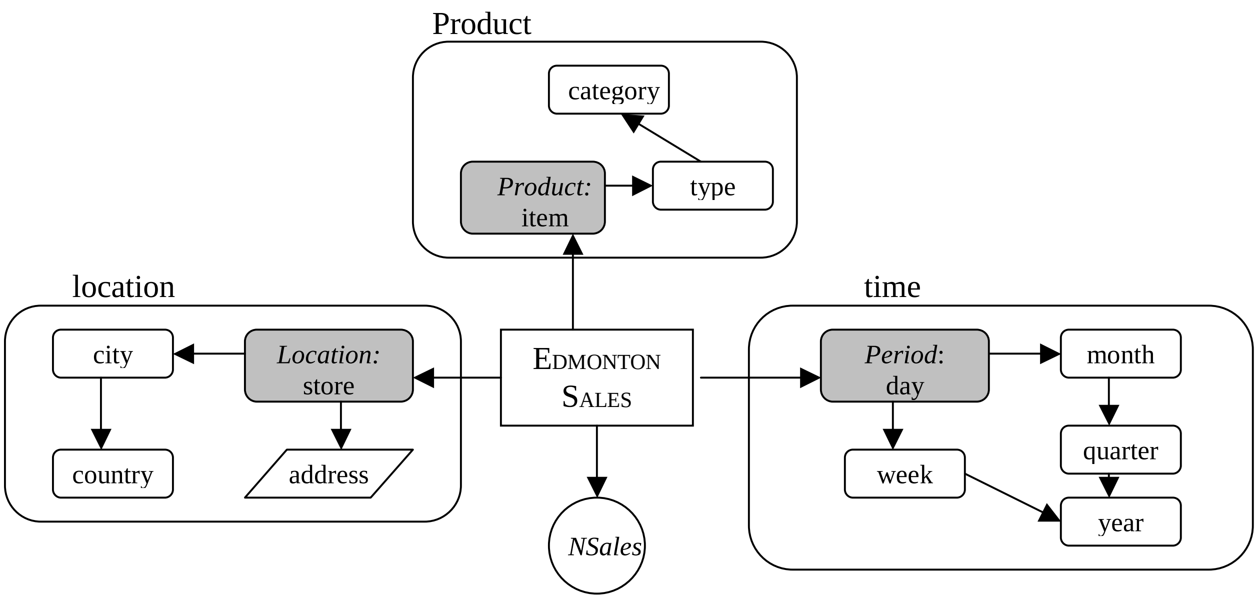


Figure 2.5 Graphical Query for EDMONTONSALES

MD model is independent of any specific implementation, but we can note it closely related to relational model. The f-table can be stored into relational tables, so it's easy to implement on the top of RDBMS.

2.4 Microsoft SQL Server Multidimensional Expressions (MDX)

The two models discussed above come from academic, and no implementation has been done based on them. Here we will take a look at a commercial software product, Microsoft SQL Server OLAP Services, which provides fast and efficient responds to user queries on multidimensional data.

How multidimensional data is stored in MS SQL Server is unclear, but they provide a query language to access it, full-fledged, highly functional expression syntax: multidimensional expressions (MDX) [Nolan99].

Since our XMDQL is mostly generated by the idea of MDX, we will discuss MDX in a lot more details. Before we introduce MDX expressions, the concepts used in MDX are almost the same as those we talked before, like cube, dimension, measure and level.

The data model is similar to data cube model, multidimensional cube with dimensions to describe category information and measures to identify the numerical values. Each dimension also contains a hierarchy of levels to view data granularly -- each level in a dimension is of a finer grain than its parent. But MDX doesn't support partial order, so there is only one way from op level to bottom level. In MDX, measures can be also viewed as dimension, special dimension.

FoodMart Sales Cube sample

The Microsoft SQL Server OLAP Services include a sample multidimensional database called FoodMart, and this is the sample data we used in our project, too. There is a Sales cube designed for the analysis of a chain of grocery stores and their promotions, customers, and products. Tables 1 and 2 outline the dimensions and measures associated with this cube.(example data comes from [Nolan99])

Table 3. Sales Cube Dimensions

Dimension name	**eelst**	**Destton**
Customers	Coutrtteor roeCtme	eorerrorrestere ustomersoourstores
utoee	utoee	utoeeoustomersus ruteereeor oo eree
eer	eer	Customereer or
rtttus	rtttus	Customermrtsttus or
rout	rout m routertmet routCteor routu teor	eroutsttreosete oortstores

	rme routme	
romotœ	e e	emeuseorromotosu ser oor eeso
romotos	romotome	etesromototttrerete se
tore	toreCoutr tore eo toreCt toreme	eorerroreret storeste outr eo t
tore e	tore ure eet	reoue stores ureeet
tore e	tore e	eostoresuseu e uermr etormroer
me	ersurtersots	meeroetesesme
er ome	er ome	omeoustomer

Table t Sales Cube easues

easue name	Destton
tes	um eroutsso
toreCost	Costooosso
torees	ueosestrstos
esCout	um erosestrstos
toreeset	ueosestrstosessostooosso
es ere	toreses sesout ssutemesure

MDX syntax

If we want a table like report for measure values on two cube dimensions, we can write a simple MDX expression like this:

```
SELECT axis specification ON COLUMNS,
axis specification ON ROWS
FROM cube_name
WHERE slicer_specification
```

Because most queries just want few dimensions projected on return cubes' axis, so MDX use `COLUMNS` to indicate first dimension, `ROWS` to state second dimension. If more than tow dimensions returned, the named axis would `PAGES`, `CHAPTERS` and, finally, `SECTIONS.` If you desire more generic axis terms over the named terms, you can use the `AXIS(index)` naming convention, with zero-based index referred to the axis. The axis specification is used to select members from cube dimensions. The slicer specification on `WHERE` clause is used to define the slice of the cube to be viewed.

The simplest form of an axis specification is taking the MEMBERS of the required dimension, including those of the special Measures dimension. For example, if we want to get all recorded measures for each store, the MDX expression is:

```
SELECT Measures.MEMBERS ON COLUMNS,
[Store].MEMBERS ON ROWS
FROM [Sales]
```

It will display all measure values for the stores hierarchy for each stores and every defined summery level.

We can also only select a single member of a dimension, like this expression:

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store State].[CA], [Store].[Store State].[WA]} ON ROWS
FROM [Sales]
```

It queries the measures for the stores summarized for the states of California and Washington. If we want to query measures for cities in these two states, we would query the CHILDREN of the required members:

```
SELECT Measures.MEMBERS ON COLUMNS,
{[Store].[Store Region].[CA].CHILDREN,
   [Store].[Store Region].[WA].CHILDREN} ON ROWS
FROM [Sales]
```

So we have two functions, MEMBERS and CHILDREN. MEMBERS function returns the members for the specified dimension or dimension level, and CHILDREN function returns the child members for a particular member within the dimension.

And DESENDANT function can drill down to a lower level within a dimension. Besides, MDX provides many functions to navigate the hierarchy, to calculate members and to analysis tie period.

Slicer specifications

In the `WHERE` clause, the slicer specification summarizes the slice of the cube to be viewed. And it is often used to select measures to be viewed because measures are together another dimension. For instance, we want to get sales average for the stores at state level, cross-referenced against the store type, the expression is:

```
SELECT {[Store Type].[Store Type].MEMBERS} ON COLUMNS,
{[Store].[Store State].MEMBERS} ON ROWS
FROM [Sales]
WHERE (Measures.[Sales Average])
```

Another example, if w only want the sales averages for the year 1999, the `WHERE` clause would be written as:

```
WHERE (Measures.[Sales Average], [Time].[Year].[1997])
```

It is important to note that slicing is not the same as filtering. Slicing does not affect selection of the axis members, but rather the values that go into them. This is different from filtering, because filtering reduces the number of axis members.

3. Other Related Technologies

3.1 XML

"The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web."[W3C00]

The Extensible Markup Language (XML) is descriptively identified in the XML 1.0 W3C Recommendation as "an extremely simple dialect [or 'subset'] of SGML"[W3C98] the goal of which "is to enable generic SGML to be served, received, and processed on the Web in the way that is now possible with HTML,"[W3C98] for which reason "XML has been designed for ease of implementation, and for interoperability with both SGML and HTML."[W3C98]

Inherited from SGML, XML is a method for putting structured data in a text file. The XML documents are self-described, so both human and machine can understand. And most important, XML documents are platform independent, language independent, so once written, they can be used everywhere. That's why XML is often used for transferring data between different heterogeneous systems.

3.2 CORBA

CORBA is an acronym for Common Object Request Broker Architecture, defined by Object Management Group (OMG). CORBA is "an open, vendor-independent architecture and infrastructure that computer applications use to work together over networks"[OMG01]. CORBA-based programs from any vendor, on any computer, operating system, programming language, and network, can communicate with each other. Because of its vendor-independent feature, CORBA is often chosen as the middleware for large enterprise.

CORBA applications are composed of *objects*, individual units of running software that combine functionality and data. IDL (Interface Definition language) is used to define the interface of the object requests. The created file that defines the object export methods and is written in IDL does not contain the actual implementation of the algorithms. The implementation of the interface is defined using Java or other language outside of the IDL and build on the framework created by some IDL converter.

3.3 SOAP

Remote objects like CORBA objects can give a program almost unlimited power over network, However, in Internet, most firewalls block non-HTTP request, so we have to find another way to do distributed computing on Internet.

SOAP (Simple Object Access Protocol) defines the use of XML and HTTP to access services, objects, and servers in a platform-independent manner. It "provides a simple and lightweight mechanism for exchanging structured and typed information between peers in a decentralized, distributed environment using XML."[W3CSOAP00]

Like CORBA, SOAP is a protocol that acts as the glue between heterogeneous software components. If developer can agree on HTTP and XML, SOAP offers a mechanism for bridging competing technologies in a standard way.

SOAP consists of three parts[W3CSOAP00]:

- The SOAP envelope construct defines an overall framework for expressing **what** is in a message; **who** should deal with it, and **whether** it is optional or mandatory.
- The SOAP encoding rules defines a serialization mechanism that can be used to exchange instances of application-defined datatypes.
- The SOAP RPC representation defines a convention that can be used to represent remote procedure calls and responses.

3.4 Jakarta Tomcat

Jakarta is the name of an project committed to open source development. Apache Software Foundation currently oversees the project. The group in responsible for developing a number of deferent produces, one being Tomcat. Tomcat is a web server extension that implements Sun Microsystems's Java Servlet 2.2 API specification. Servlets are written using only the Java programming language. Servlet allows a web server to be extended such that it can provide additional services. One such service is the creation of dynamic web pages that are returned to the web browser. Servlets are associated with a particular URL.

A servlet is used to extend the capability of a web server. The web server receives a GET connection request from a specific web browser. The GET request can contains a specific URL and html action request is received by the web server and invokes the Java servlet code corresponding to the URL and html action request. This provides the web server extensions. The servlet contains the ability to respond to the request. A service method within the HttpServlet API is called which hands off the client request according to the type of the HTTP request.

By extending the HttpServlet Java API, doGet and doPost methods are defined. The doGet method responds to HTTP GET request, which is normal, a request for a web page. The doPost method responds to the HTTP POST command, which is produced as a response of a button press where the button resides on the web pages of the client's web browser in order to submit information to the web server.

Servlets allow the web server to create dynamic web pages based on some user input. In this manner servlets are similar to CGI. The POST and GET HTTP request allows parameters to be passed from the web client to servlet via the web server, the servlet once invoked responds to the HTTP request. Using servlets to create dynamic web pages, the response is a web page generated by the servlet. The web page is written to Java PrintWriter stream object that hands the servlet generated web page back to the client.

Servlets are written in Java and overcome some of the drawbacks of CGI. Java allows for the portably of the byte code across many platforms where CGI written in the C, C++ programming language must be recompiled in order to run on various platforms. Unlike C, C++, the Java language does not suffer as much from the same issues. Java contains a build in garbage collection method that frees memory still allocated by references that have gone out of scope. Memory management in C, C++ CGIs could easily contain memory leaks. C, C++ CGI sometimes provided for possible security breaches because bounds checking were not done property on the values submitted by the web browser. This led to root exploits on the system. Java utilizes dynamically allocating objects (vectors, strings, et) to handle varying sizes of inputs to the servlets.

3.5 Xerces Java Parser

In our implementation, we use Apache Xerces Java parser 1.3 to parse XML documents. It supports XML 1.0 recommendation and contains advanced parser functionality, such as XML Schema, DOM Level 2, and SAX 2.0, in addition to supporting the industry-standard DOM Level 1 and SAX 1.0.

Xerces Java parser provides standard W3C DOM API and javax.xml.parsers API, and we use it as a standard XML parser, so it can be changed to any other XML parser for Java, such as those made by SUN, IBM or Oracle.

3.6 Apache SOAP

For SOAP (Simple Object Access Protocol) usage, we choose Apache SOAP implementation. It is based on the IBM SOAP4J implementation. IBM donated their code to the Apache Software Foundation, and the open source community quickly embraced the project and has been working nonstop on bug fixes, enhancements, and new functionality.

The Apache SOAP service can be running under any servlet containers, and it works very well with Jakarta Tomcat, although it’s not easy to set up the server and try your first SOAP program.

4. Virtual Data Warehouse System Design

4.1 System Architecture

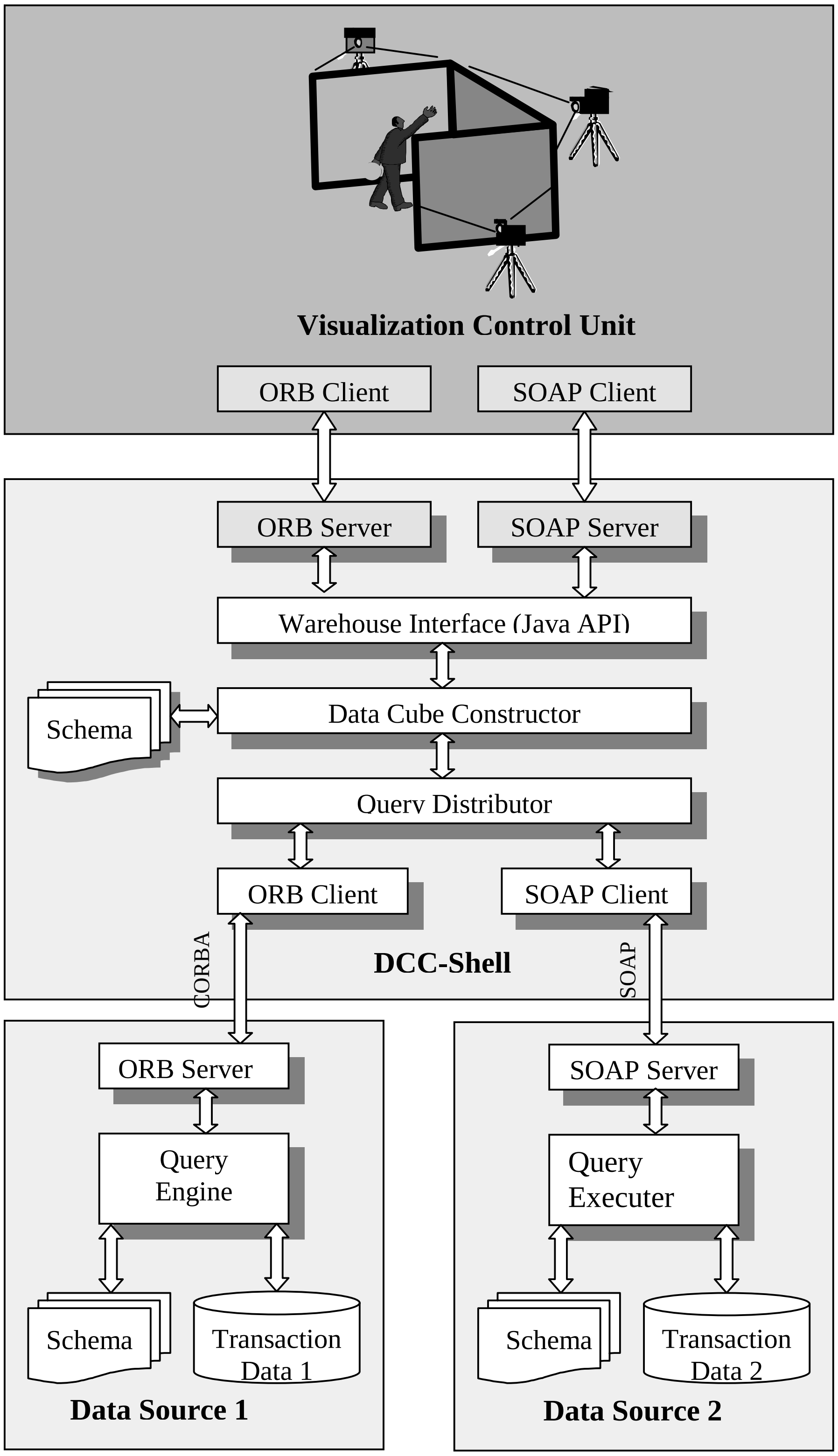


Figure 4.1. Virtual Data Warehouse Architecture

The Virtual Data Warehouse (VDW) is a conceptualization of a centralized data warehouse that includes a set of distributed data sources and a shell (DCC-Shell) that is responsible for managing and querying these sources. The DCC-Shell does not store any actual transaction data. Instead, these transaction data are left on the distributed sites containing them while the DCC builds and updates a global multidimensional model of the available dimensions and measures, hence the name “virtual warehouse”. This approach provides the VDW clients constantly updated views of any constructed cubiod in a manner that makes the source distribution transparent to the user. Although the DCC-Shell does not store raw data, it maintains a pool of meta-data (cube schema) that is synchronized with all data sources. This global schema is prepared when constructing the VDW, and if any parts are changed, all sites must be updated to avoid inconsistencies. Although this approach is hard to maintain, it’s much easier to implement. Besides meta-data, DCC-Shell also stores a resource allocation table that includes information pertaining to the location, data organization, and the communication method of each data sources. All the meta-data and resource allocation data are written in XML format, so it is easy to understand and maintain, therefore makes the whole system extensible and flexible.

Virtual data warehouse, like traditional warehouse, or database, must provide services, and we call them queries. Right now there are three classes of query functions available to a client.

We classify the query into three kinds, warehouse query, cube schema query and cube data query.

- **Warehouse query**: user requests basic info about the warehouse, and results are warehouse name, description, how many data cubes it has, the size of each cube. It is useful when a client first accesses the VDW to get brief idea of it.

- **Cube schema query**: user asks for the meta-data of one specific cube, and results are cube description, measurement, dimension, etc.

- **Cube data query**: It is used to obtain an entire N-dimensional cube or any subset of it. This is particularly useful in applications such as the VCU that handles only one 3D cube per visualization session. Just like meta-data storage, all query requests and responds are in XML format.

4.2 Schema Repository

“Data warehouses are based on a multidimensional data model. This model views data in the form of a data cube.” A cube is defined by dimensions and facts. In our system, a cube has schema part and fact data part, and how to design cube schema in XML becomes one important issue.

First, a cube schema has name property and sub elements of measures and dimensions, which are sets of measures and dimensions:

```
<!ELEMENT CubeSchema (Measures, dimensions)>
<!ELEMENT Measures (Measure*)>
<!ELEMENT Dimensions (Dimension*)>
```

Each Measure element has name, aggregationFunction as properties, and Title, DataType sub-elements:

```
<!ELEMENT Measure (Title, DataType)>
<!ATTLIST Measure name NMTOKEN >
<!ELEMENT Title (#PCDATA)>
<!ELEMENT DataType (#PCDATA)>
```

For example, we have following measures:

```
 <Measures>
  <Measure name="Unit_Sales" aggregationFunction="SUM">
   <Title>Unit Sales</Title>
   <DataType>double</DataType>
  </Measure>
  <Measure name="Store_Cost" aggregationFunction="SUM">
   <Title>Store Cost</Title>
   <DataType>double</DataType>
  </Measure>
  <Measure name="Store_Sales" aggregationFunction="SUM">
   <Title>Store Sales</Title>
   <DataType>double</DataType>
  </Measure>
  <Measure name="Sales_Count" aggregationFunction="Count">
   <Title>Store Cost</Title>
   <DataType>double</DataType>
  </Measure>
 </Measures>
```

Next comes the most difficult part, dimensions. “The dimensions of a cube represent distinct categories for analyzing business data. Categories such as time, geography, or product line breakdowns are typical cube dimensions.”

Typically, a dimension is organized into a hierarchical data structure, and may has partial or total ordering. To simplify the problem, we design the dimension with total ordering, and save partial ordering info as properties of dimension units.

Here we try to define dimension by example of store. The concept hierarchy of store location can have many levels of categories, such as continent, country, region and city, as in the following figure:

So we can write down the dimension of store:

```
   <Dimension name=”Store” allLevel="yes" allCaption="All Store">
    <Description>The store schema</Description>
    <Levels number=”5”>
     <Level name=”Continent” Title=”Store Continent”/>
     <Level name=”Country” Title=”Store Country”/>
     <Level name=”Region” Title=”Store Region”/>
     <Level name=”City” Title=”Store City”/>
     <Level name=”Store” Title=”Store Name” type=”base”>
      <Property name=”Store_Manager” type=”String”/>
     </Level>
    <Levels>

    <Unit name=”N_America” >
     <Unit name=”USA”>
      <Unit name=”West”>
       <Unit name=”Seattle”>
        <BaseUnit name=”Store1” baseID=”1”>
         <Property name=”Store_Manager”>Bill Gates</Property>
        </BaseUnit>
       </Unit>
      </Unit>
      <Unit name=”Central”>
       <Unit name=”Denver”>
        <BaseUnit name=”Store2” baseID=”2”/>
       </Unit>
      </Unit>
      <Unit name=”East”>
       <Unit name=”Boston”>
        <BaseUnit name=”Store3” baseID=”3”/>
       </Unit>
      </Unit>
     </Unit>
     <Unit name=”Canada”>
      <Unit name=”BC”>
       <Unit name=”Vancouver”>
        <BaseUnit name=”Store4” baseID=”4”/>
       </Unit>
      </Unit>
      <Unit name=”Alberta”>
       <Unit name=”Edmonton”>
        <BaseUnit name=”Store5” baseID=”5”/>
       </Unit>
      </Unit>
      <Unit name=”Ontario”>
       <Unit name=”Toronto”>
        <BaseUnit name=”Store6” baseID=”6”>
         <Property name=”Store_Manager”>Stockwell Day</Property>
        </BaseUnit>
       </Unit>
      </Unit>
     </Unit>
    </Unit>
    <Unit name=”Europe”>
    </Unit>
   </Dimension>
```

We can also write down another dimension for product:

```
<Dimension name=”Product” allLevel="yes" allCaption="All Product">
 <Description>The product schema</Description>
 <Levels number=”3”>
  <Level name=”Category” Title=”Product Category”/>
  <Level name=”Type” Title=”Product Type”/>
  <Level name=”Product” Title=”Product Name” type=”base”/>
 <Levels>

 <Unit name = “Office” >
  <Unit name = “Computer”>
   <BaseUnit name = “IBM” baseID = “0”/>
```

```
       <BaseUnit name = “Compaq” baseID = “1”/>
       <BaseUnit name = “Apple” baseID = “2”/>
      </Unit>
      <Unit name = “Fax”>
       <BaseUnit name = “Panasonic” baseID = “3”/>
       <BaseUnit name = “Brothers” baseID = “4”/>
      </Unit>
      <Unit name = “Copier”>
       <BaseUnit name = “Cannon” baseID = “5”/>
       <BaseUnit name = “Xerox” baseID = “6”/>
      </Unit>
     </Unit>
     <Unit name = “Appliance”>
      <Unit name = “Kitchen”>
       <BaseUnit name = “Brown” baseID = “7”/>
       <BaseUnit name = “Sharp” baseID = “8”/>
      </Unit>
      <Unit name = “House”>
       <BaseUnit name = “GE” baseID = “9”/>
      </Unit>
     </Unit>
     <Unit name = “Entertainment”>
      <Unit name = “TV”>
       <BaseUnit name = “27” baseID = “10”/>
       <BaseUnit name = “37” baseID = “11”/>
      </Unit>
      <Unit name = “Stereo”>
       <BaseUnit name = “100W” baseID = “12”/>
       <BaseUnit name = “500W” baseID = “13”/>
      </Unit>
     </Unit>
    </Dimention>
```

Above two examples are common dimensions, and we also have some special dimensions, such as time. Time’s concept hierarchy can be presented as partial ordering or total ordering.

In figure 1.2, day has two ways to roll up, either to month or to week, so this hierarchy cannot be presented by tree structure. But if we put week into day level as a property, we will make it a total order. Here is an example of Time dimension schema in XML:

```
<Dimension name="Time" special="time"
             allLevel="no" allCaption="All Level">
   <Description>Time dimension for Cube</Description>
   <Levels number="3">
    <Level name="Year" Title="Year"/>
    <Level name="Quarter" Title="Quarter"/>
    <Level name="Month" Title="Month"/>
    <Level name="Day" Title="Day" type="base">
     <Property name="Day_of_Week" type="String" />
    </level>
   </Levels>
   <Unit name="1997">
    <Unit name="Q1">
     <Unit name="January">
      <Unit name="1" baseID="367"> <Property>Wednesday</Property></Unit>
      <Unit name="2" baseID="368"> <Property>Thursday</Property></Unit>
      <Unit name="3" baseID="369"> <Property>Friday</Property></Unit>
………
```

```
      </Unit>
    </Unit>
  </Diemnsion>
```

4.3 XML Multi-Dimensional Query Language (XMDQL)

We propose an XML-based query language, XMDQL, to interact with the VDW in order to manage and access the available data. The concept of a special multidimensional query language was first proposed (still not finalized) by Pilot software [16] as an industry standard. In OLAP terminology this type of query is equivalent to slicing and dicing the data cube. The result of executing an XMDQL query is a cell, a two-dimensional slice, or a multidimensional sub-cube. XMDQL provides functionality similar to that of Microsoft's MDX (Multidimensional Expressions), but it is formatted in XML and the result is also XML document.

basic format

To specify a cube, XMDQL must contain information about following subject:

- The virtual cube that query is on.
- Dimensions projected in result cube.
- Slices in each dimension to present and sort order.
- The members from a nonprojected dimension on which data will be filtered for members from projected dimensions.

XMDQL has this basic forms:

```
<XMDQL>
  <SELECT>
      project dimensions and  slices
  </SELECT>
  <FROM>
    witch cube to query
  </FROM>
  <WHERE>
     filtering constrains
  </WHERE>
</XMDQL>
```

examples

For example, we have a virtual cube with four dimensions, Store, Time, Product, Customer, and we want to see sales for USA for office product and for each quarter in 1997.

Table 5. Sales for USA

	Computer	Fax	Copier
Quarter Q1	4356	342	56456

Quarter Q2	556	945	234
Quarter Q3	8754	656	3324
Quarter Q4	456	786	4334

In this example, the product and time dimension are projected dimensions, and each has one slice. Store dimension is used in WHERE part as filter.

```
<XMDQL>
  <SELECT>
    <Measure name=”Store_Sales”>
    <Axis dimension=”Product”>
      <Slice type=”mono” title=”${name}”>
        <Path>Office.*</Path>
      </Slice>
    </Axis>
    <Axis dimension=”Time”>
      <Slice type=”mono” title=”Quarter ${name}”>
        <Path>1997.*</Path>
      </Slice>
    </Axis>
  </SELECT>
  <FROM>
    <Cube name=”AllElectronics”/>
  </FROM>
  <WHERE>
    <Condition dimension=”Store”>
      <Path>N_America.USA</Path>
    </Condition>
  </WHERE>
</XMDQL>
```

In XMDQL, we introduce **`path`** in Slice, to indicate how to retrieve data. The `path=”Office.*”` means show all children of Office, so the generated set is {Computer, Fax, Copier}. Therefore, in the axis of dimension Product, only the sales of computer, fax and copier are presented. In time axis, `path=”1997.*”`, and below 1997, there are 4 children, Q1, Q2, Q3 and Q4. In WHERE part, the condition has one dimension filter, meaning only sales in USA stores are retrieved.

The path is a dot separated dimension unit name stream, and shows a route in the concept hierarchy. The ‘*’ means all children of the unit. If we want to see sales on all product, we set path=”*”. Path uses ‘{}’ to query on level. For example, if we want to see sales on each product type, the path is set to “{Type}”, so the return set of dimension Product is {Computer, Fax, Copier, Kitchen, House, TV, Stereo}. See **Path Syntax** for formal definition of path.

The Slice element sets the member or members of dimension to project on axis. We can select more than one slice in one dimension. For example, we want the time axis has these units:
January, February, March, Q2, Q3, October, November, December.
The query can be as following:

```
<Axis dimension=”Time”>
  <Slice type=”mono” title=”1997 ${name}” >
```

```
    <Path>1997.Q1.*</Path>
  </Slice>
  <Slice type=”mono” title=”1997 Quarter %{name}” >
    <Path>1997.Q2</Path>
    <Path>1997.Q3</Path>
  </Slice>
  <Slice type=”mono” title=”1997 ${name}” >
    <Path>1997.Q4.*</Path>
  </Slice>
</Axis>
```

We can also customize new units in slice. For example, we want see total sales of USA and Canada, so Slice is:

```
   <Slice type=”sum” title=”USA and Canada”>
     <Path>N_America.USA</Path>
     <Path>N_America.Canada</Path>
   </Slice>
```

The `type` attribute of `Slice` can be `mono`, `sum`, etc. `sum` means aggregating the total data in all paths and displaying as one projected unit.

Path syntax

Path is used to indicate the units projected on one axis. We define a path is consisted of several path nodes separated by “.” Tokens.

```
Path ::= PathNode   ( SEPARATE_TOKEN  PathNode ) *
SEPARATE_TOKEN ::=  ‘.’
PathNode ::= ( ‘{‘ LevelName ‘}’ )? ( UnitName )? ( ‘[‘ Condition
‘]’ )?
LevelName ::= name string of level.
UnitName ::= name string of unit.
Condition ::= Boolean expression on properties of unit.
```

Examples:
`1997.{Day}` : see data of each day of 1997.
`1997.{month}.13` : see data of each month’s 13.
`1997.{month}.13[day_of_week=”Friday”]` : see data of each 13 Friday in 1997.
`{Store}[manager=”Tom”]` : see data of stores whose manager is Tom.

XMDQL DTD

```
<!ELEMENT XMDQL (SELECT, FROM, WHERE?)>
<!ELEMENT SELECT (Measure+, Axis+)>
<!ELEMENT FROM (Cube)>
<!ELEMENT WHERE (Condition+)>

<!ELEMENT Measure EMPTY>
<!ATTLIST Measure name CDATA>

<!ELEMENT Axis (Slice+)>
<!ATTLIST Axis dimension CDATA>
```

```
<!ELEMENT Slice (Path+)>
<!ELEMENT Slice type “mono|sum” “mono”
                title CDATA >

<!ELEMENT Path (#PCDATA)>

<!ELEMENT Cube EMPTY>
<!ELEMENT Cube name NMTOKEN>

<!ELEMENT Condition (Path+)>
<!ATTLIST Condition dimension CDATA
                    type “include|exclude” “include” >
```

4.4 Query Distribution and Execution

According to figure 2.1, the query is received by DCC-Shell interface, and sent to Data Cube Constructor, then sent to Query Distributor. The Query Distributor analyzes the query and finds which data marts contain the cube data, then distributes the query to them.

The question is how to store the cube data distribution information so distributor can easily figure out the Data Marts to communicate.

Because we assume the cube schema is stored in each site with same copies, only fact data are distributed. And another assumption is that we distribute data on only one dimension. For example, the sales data of USA stores are in Data Mart 1, and Canada in Data Mart 2.

So we can save the distribution info in XML like:

```
<Distribution dimension=”Store”>
 <Component path=”N_America.USA”
           mart=”DataMart1”>USA sales data</Component>
 <Component path=”N_America.Canada
           mart=”DataMart1”>Canada sales data</Component>
</Distribution>
```

Suppose we have distribution component set $D_{dist} = \{C_1, C_2, \ldots, C_n\}$, each C_i related to a branch of concept hierarchy tree, and there is no overlap between them. And we can also define dimensions in SELECT or WHERE part of XMDQL as set D_{query}, for each dimension, $D_j=\{s_{j1}, s_{j2}, \ldots, s_{jmj}\}$, s is slice. For $D_j=D_{dist}$, we check if C_i embrace one slice, add the data source i. If no Dj equals to $D_{dist,}$ add all data sources.

In the distribution algorithm, we should also consider function slice and complex WHERE conditions.

Assume the XMDQL query has no function slice and no complex condition in WHERE, and the projected dimensions are sliced into display units. For each unit, let set U include all base unit ID below it, so we get :

PD_1 : U_{11}, U_{12}, U_{13}, …, U_{1m1}
PD_2 : U_{21}, U_{22}, U_{23}, …, U_{2m2}
…
PD_n : U_{n1}, U_{n2}, U_{n3}, …, U_{nmn}

So every cell in result cube has baseID set {U}. And for conditions, we also have:
CD_1 : U'_{11}, U'_{12}, U'_{13}, …, U'_{1m1}
CD_2 : U'_{21}, U'_{22}, U'_{23}, …, U'_{2m2}
…
CD_n : U'_{n1}, U'_{n2}, U'_{n3}, …, U'_{nmn}

Then scan through fact table with each record. If one record’s dimension baseIDs belong to one cell’s baseID set, and obey the condition, add the measure value to this cell.

Here we suppose local data marts store cube fact data as tables. The fact tables maybe saved in database, such as DB2, Oracle. So we can create some stored procedures to retrieve data.

The algorithm for this query engine needs more study in the future.
Combining cubes from distributed data marts into result cube may be as easy as sum cell values of each cube if there is no function slice.

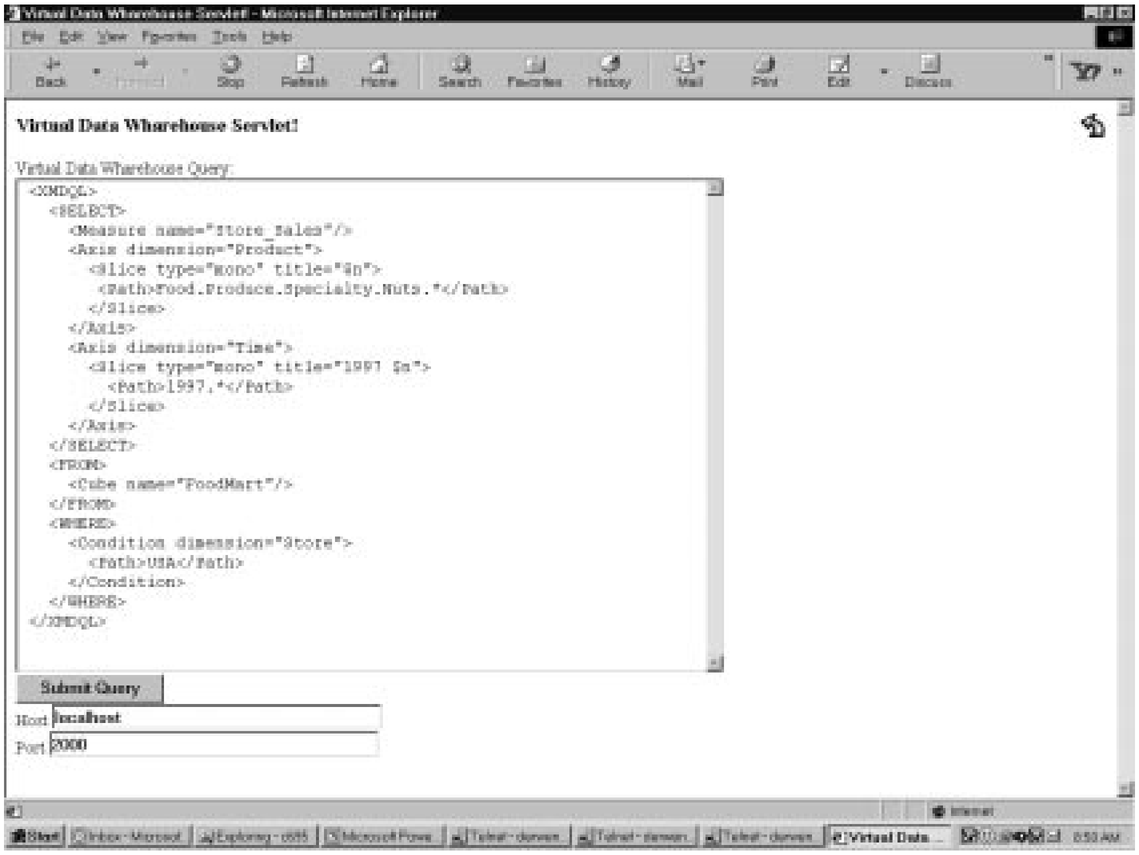
The returned cube should include axis information and measure data. Following example in table 5, we can write cube as this:

```
<CubeData>
 <Measure name=”Store_Sales”>
 <Axis>
  <Dimension name=”Time”>
   <Unit path=”1997.Q1”/>
   <Unit path=”1997.Q2”/>
   <Unit path=”1997.Q3”/>
   <Unit path=”1997.Q4”/>
  </Dimension>
  <Dimension name=”Product”>
   <Unit path=”Office.Computer”/>
   <Unit path=”Office.Fax”/>
   <Unit path=”Office.Copier”/>
  </Dimension>
 </Axis>
 <Data>
  <D1>
   <D2>43 45 567</D2>
   <D2>56 44 77</D2>
   <D2>22 87 345</D2>
   <D2>45 76 34 234</D2>
  </D1>
 </Data>
</CubeData>
```

4.5 Implementation interface

Finally, we implemented the XMDQL query engine by Java language, and set up a CORBA server as Virtual Data Warehouse. For time limit, we only set up one SOAP server as data source to store transaction data. The fact data was saved as local files and organized in a special format so XMDQL query engine could get the data quickly.

Because DIVE-ON system was not ready then, we used Jakarta Tomcat to create a web site to provide XMDQL query tools by a simple interface. User can visit the web page of this site to input XMDQL string, in XML format, and submit the query. The query was posted to servlet inside Tomcat web server, then the servlet sent the query string to our virtual data warehouse ORB server. The DCC-Shell did the query distribution and forwarded query string to another SOAP server.


Virtual Data Wharehouse Servlet! - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail Print Edit Discuss

Virtual Data Wharehouse Servlet!

Virtual Data Wharehouse Query:

```
<XMDQL>
  <SELECT>
    <Measure name="Store_Sales"/>
    <Axis dimension="Product">
      <Slice type="mono" title="$n">
       <Path>Food.Produce.Specialty.Nuts.*</Path>
      </Slice>
    </Axis>
    <Axis dimension="Time">
      <Slice type="mono" title="1997 $n">
        <Path>1997.*</Path>
      </Slice>
    </Axis>
  </SELECT>
  <FROM>
    <Cube name="FoodMart"/>
  </FROM>
  <WHERE>
    <Condition dimension="Store">
      <Path>USA</Path>
    </Condition>
  </WHERE>
</XMDQL>
```

Submit Query

Host: localhost

Port: 2000

Figure 4.2 XMDQL Query Interface

Now the interface only lets users do multidimensional data query on two dimensions, and the return values are displayed in a table format. But the VDW can handle any XMDQL query with unlimited number of dimensions (not exceeding cube dimension number). The response time is pretty good, almost as good as MS SQL OLAP server.

5. Conclusion and Future Work

In this report, we discussed the concept of data warehouse as well as data model and query language of multidimensional databases. We also designed and implemented a virtual data warehouse with java language and CORBA, SOAP techniques. The main contributions of our work are applying XML technology to data warehouse and a new multidimensional data query language XMDQL.

There are many areas of this work that offer opportunities for future work. We didn't have enough time to investigate how to implement different wrapper for different data source, such as many legacy systems, so translating XMDQL to local system query language is a big issue for further study. And many interesting topics will appear like run-time data cleaning, integration and performance. Another opportunity will be efficiently passing cube data between DCC-Shell and data sources, for right now cube data are formatted in XML document, and the size is too big for ordinary queries.

References

[**Agrawal97**] Rakesh Agrawal, Ashish Gupta, Sunita Sarawagi, *Modeling Multidimensional Databases*. ICDE 1997 http://www.almaden.ibm.com/cs/people/ragrawal/papers/md_model_rj.ps

[**Cabibbo97**] Luca Cabibbo, Riccardo Torlone, *Querying Multidimensional Databases*. DBPL 1997

[**Cabibbo98**] Luca Cabibbo, Riccardo Torlone, *A Logical Approach to Multidimensional Databases*. EDBT 1998.

[**Cabibbo00**] Luca Cabibbo, Riccardo Torlone, *Query Language for Multidimensional Data*. Rapporto tecnico InterData T4-R22, 2000

[**Chaudhuri97**] Surajit Chaudhuri, Umeshwar Dayal, *An Overview of Data Warehousing and OLAP Technology*. SIGMOD Record 26(1): 65-74 (1997)

[**Gray95**] Jim Gray, Adam Bosworth, et al. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals. Microsoft Technical Report MSR-TR-95-22 Nov. 1995

[**Hammond99**]Mark Hammond, *Survey traces huge growth in data warehouse market*. EWeek Dec. 13, 1999.

[**Holland00**] Paul Holland, *Virtues of a Virtual Data Warehouse*. EarthWeb.com March 2000.

[**Inmon92**] W.H. Inmon, *Building the Data Warehouse*. John Wiley, 1992.

[**Inmon96**] W.H. Inmon, *Virtual Data Warehouse: The Snake Oil Of The Nineties*. Data Management Review, April, 1996.

[**Jordan95**] Sara Jordan and John Wheeler, *The Virtual Data Warehouse*. http://www.ornl.gov/cim/vdwtxt.htm Dec. 19, 1995

[**Kimball96**]Ralph Kimball, *The Data Warehouse Toolkit : Practical Techniques for Building Dimensional Data Warehouses*. John Willy & Sons 1996.

[**Nolan99**] Carl Nolan, *Introduction to Multidimensional Expressions (MDX).* Aug. 1999

[OMG01] *CORBA Basic*, http://www.omg.org/gettingstarted/corbafaq.htm, April, 2001

[**Orr96**] Ken Orr, *Data Warehousing Technology*. 1996

[**Sakaguchi96**] Torus Sakaguchi, Mark Frolick, *A Review of the Data Warehousing Literature.* http://www.nku.edu/~sakaguch/dw-web.htm 1996

[**Trujillo99**] Juan Carlos Trujillo, *The GOLD model: An Object Oriented multidimensional data model for multidimensional databases*. ECOOP Workshop for PhD Students in OO System 1999: 24-30 http://www.comp.lancs.ac.uk/computing/users/marash/PhDOOS99/Paper17_Juan_Trujillo.pdf

[W3CSOAP00] Don Box, et al. *Simple Object Access Protocol (SOAP) 1.1* http://www.w3.org/TR/SOAP/ may, 2000

[**W3CXML00**]*Extensible Markup Language (XML)* http://www.w3c.org/xml, 2000

[**W3CXML98**] Tim Gray, et al. *Extensible Markup Language (XML)1.0 (Second Edition)* http://www.w3.org/TR/REC-xml, Oct. 1998