Distributed Data Warehouse Management in a Parallel Environment Using Compressed Relational Structure

by

Fazlul Hasan Siddiqui
Roll No. 040405047P

A thesis submitted to the Department of Computer Science and Engineering in partial fulfillment of the requirements for the degree of MASTER OF SCIENCE IN COMPUTER SCIENCE AND ENGINEERING

Department Of Computer Science and Engineering

BANGLADESH UNIVERSITY OF ENGINEERING AND TECHNOLOGY, DHAKA

February, 2007

The thesis **"Distributed Data Warehouse Management in a Parallel Environment Using Compressed Relational Structure"**, submitted by Fazlul Hasan Siddiqui, Roll No. 040405047P, Session: April 2004, to the Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, has been accepted as satisfactory for the partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Science and Engineering) and approved as to its style and contents for the examination held on 26th February, 2007.

Declaration

It is hereby declared that the work presented in this thesis or any part of this thesis has not been submitted elsewhere for the award of any degree or diploma, does not contain any unlawful statements and does not infringe any existing copyright.

Signature

[illegible]

(Fazlul Hasan Siddiqui)

Table of Content

List of Figures

List of Tables

Acknowledgement

First of all, I want to express my sincere gratitude to my supervisor, Dr. Abu Syed Md. Latiful Hoque, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology, Dhaka for making the research as an art of exploiting new ideas and technologies on top of the existing knowledge in the field of the database and the data compression. He provides me moral courage and excellent guideline that make it possible to complete the work. His profound knowledge and expertise in this field provide me many opportunities to learn new things to build my carrier as a researcher.

I would like to acknowledge Dr. Gagan Agrawal, Associate Professor, Dept. of Computer Science and Engineering, Ohio State University, United States for providing his paper on parallel data cube construction. Also I would like to thank Yannis Sismanis, Dept. of Computer Science, University of Maryland, College Park, United States for providing his paper on shrinking the petacube. Then I am thankful to Philip Greenspun and Jim Gray from Microsoft for supporting enough online Microsoft Technical report on data warehousing.

I also express my gratitude to Professor Dr. Md. Shahid Ullah, Head of the Department of Computer Science and Engineering, DUET, Gazipur for providing me enough lab facilities to make necessary experiments of my research. I am also grateful to my colleagues and relatives for encouraging me to continue my research.

Abstract

Data warehousing is a key technology in everyday activity, which usually contains historical data derived from transaction data, but it can include data from other sources. The objective of a data warehouse is to provide analysts and managers with strategic information underlying the business to consolidate data from several sources. Unfortunately, the emergence of e-application has been creating extremely high volume of data that reaches to terabyte threshold. The conventional data warehouse management system is costlier in terms of storage space and processing speed, and sometimes it is unable to handle such huge amount of data. As a result, queries and analyses are becoming more complex and time consuming. Therefore, there is a crucial need for the new algorithms and techniques to store and manipulate these data.

Parallel and distributed data warehouse architectures have been evolved to support online queries on massive data in a short time. The database compression can be used for scalable storage and faster data access. In this thesis, we have presented a compression-based distributed data warehouse architecture for storage of warehouse data, and support online queries efficiently. We have achieved a factor of 25-30 compression compared to conventional SQL server data warehouse.

The main computational component of data warehouse is the generation and querying on the data cube. Our algorithm generates data cube directly from the compressed form of data in parallel. The reduction in the size of data cube is a factor of 30-45 compared to existing methods. The response time has also been significantly improved. These improvements are achieved by eliminating the suffix and prefix redundancy, virtual nature of the data cube, direct addressability of compressed form of data and parallel computation. Experimental evaluation shows the improved performance over the existing systems.

Chapter 1 Introduction

Data Warehouse is a subject oriented, integrated, non-volatile and time-variant collection of massive data in support of management's decisions. A data warehouse environment includes an extraction, transportation, transformation, and loading (ETL) solution, an online analytical processing (OLAP) engine, client analysis tools, and other applications that manage the process of gathering data and delivering it to business users. Parallel and distributed architectures of this data warehouse have been evolved to support online queries on massive data in a short time. Data cube construction is a frequently used operation in data warehouse. This operation requires computation of aggregations on all possible combinations of each dimension attribute.

1.1 Problem Definition

As warehouse data grows exponentially, storage and querying on this data still remain a problem. Today's warehouses are approaching several terabytes of data with a broader mix of query types, which is not supported by the conventional archiving of data. As a result, this increased data put a number of strains on physical resources, IT respects and management spectrum. To overcome this problem, research is done on data warehouse to meet the local autonomy and expandability in a distributed environment. Parallel execution dramatically reduces response time for data-intensive operations on large databases typically associated with decision support systems (DSS) and data warehouses. Simply expressed, parallelism is the idea of breaking down a task so that, instead of one process doing all of the work in a query, many processes do part of the work at the same time.

1.2 Aim and Objective of the Thesis

Performance of the large scale data warehouse can be improved using compression based distributed warehouse. A number of compression techniques on centralized large database have recently been proposed [1, 2, 3, 4]. In contrast, a number of distributed warehouse architectures [5, 6, 7, 8, 9, 10] have been put forward without the application of compression methods. Integrating these two technologies, this work exhibits a complete solution to a compression based distributed data warehouse management in a parallel environment, which belongs to the first part of our contribution, and we named it as 'Distributed Multi-Block Vector Architecture (DMBVA)'.

Next, Pre-computation of the data cube is very critical to improve the response time of OLAP queries in data warehouse. However, as the size of the warehouse grows, this pre-computation becomes a significant performance bottleneck, because the required computation cost grows exponentially with the increase of dimensions. So it is natural to consider parallel architecture for this process. Therefore, this thesis demonstrates virtual data cube construction directly from the compressed and distributed data warehouse in parallel on a shared nothing multiprocessor system. We named it as 'Parallel Construction of Virtual Data Cube (PCVDC)'. Here the data cube is virtual in the sense that there is no direct storage of dimensional values in the data cube, instead only the block number and corresponding measure value are stored.

Additionally optimization techniques are applied onto the data cube for better performance, where suffix redundancy is eliminated, and therefore the resultant cube drastically reduced to a highly condensed cube. Therefore, the specific aims and possible outcome of this work are:

- Reducing storage cost using compressed representation of data with direct addressability.
- Spread the processing load across multiple servers using shared nothing multiprocessor system.
- Virtual data cube construction directly from the compressed and distributed data warehouse in parallel.
- Supporting an upper bound of construction and query response time from the constructed data cube.
- Concurrent online access by multiple users and improved performance in decision queries.
- Incorporating optimization techniques onto the data cube for better performance.

1.3 System Overview

The complete system can be divided into four parts. Therefore the overall system is adjusted into a four tier architecture which is shown in Fig. 1.1. In the tier one, data is collected from the external source of operational data, which are then extracted, transformed, loaded, refreshed and stored into the data warehouse server. Next with the intension of compressing the warehouse data, we apply our Distributed Multi-Block Vector Architecture (DMBVA) in tier two to store the relational schema in a compressed and distributed format. DMVBA is an extended modification of CMBVS [4]. Here for successful storage management, the relational warehouse is compressed and distributed over multiple column servers. Each column server is associated with a domain dictionary, a large compressed vector and an index structure. Application server synchronizes among those column servers. The domain dictionary consists of set of lexemes with corresponding token values. It is updated dynamically whenever facing a new lexeme. A compressed vector represents a single column of relational database. Here only the tokens of the corresponding lexemes are stored and hence compressed.

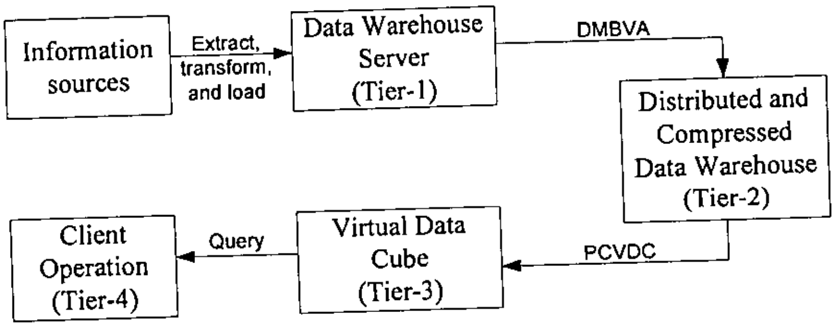

Information sources
Extract, transform, and load
Data Warehouse Server (Tier-1)
DMBVA
Distributed and Compressed Data Warehouse (Tier-2)
PCVDC
Virtual Data Cube (Tier-3)
Query
Client Operation (Tier-4)

Fig.1.1: Overall system architecture

Next in tier three we have generated the data cube directly from the compressed vector in parallel. This procedure is named as ‘Parallel Construction of Virtual Data Cube (PCVDC)’ and it belongs to the main focus of the work. A schedule tree which is generated from a cuboid lattice by solving a minimum spanning tree problem, guide all through the data cube construction procedure. For the sake of optimality issue, prefix and suffix redundancy in the data cube are identified in our approach and eliminated as well. Finally the clients do their query operation from the virtual data cube with the help of the query manager, which is synchronized by the application server.

1.4 Thesis Outlines

In chapter 2, a review of the literature is presented. There, detailed discussion about different data warehouse architectures, warehouse compression and distribution techniques, shared nothing multiprocessor system, data cube generation procedures and different query operations are depicted. We have considered the CMBVS [4] as the basis of our compression method with some modifications.

Chapter 3 describes our compressed relational architecture – DMBVA, with detailed analysis of its components – domain dictionary, compressed vector and index structure. Then the most highlighted part – 'Parallel Construction of Virtual Data Cube (PCVDC)' algorithm with the data cube file format is exposed. The optimization technique is presented in the subsequent section. Then the detailed architecture of the query operation is depicted. The details analytical description of the components of the entire system is also given.

Chapter 4 contains details of the experimental work that has been carried out and the discussions on the experiment. Here, various comparisons are made on the basis of storage space, construction and query response time against other related work. The experimental evaluation shows the improved performance over the existing systems.

Chapter 5 presents the comments on the overall system. It also renders the suggestions for the future research.

Chapter 2
Literature Review

A data warehouse is a relational database that is designed for query and analysis rather than for transaction processing. It is a collection of diverse data having the properties - subject oriented, integrated, time-varying, and non-volatile. Data warehouse is necessary to support the management's decision-making process. It has a collection of tools for cleansing, integrating, querying, reporting, analysis, data mining, monitoring, and administering data.

- **Subject-oriented:** Warehouse data is organized around major subjects, such as customer, product, sales. It focuses on the modeling and analysis of data for decision makers, not on daily operations or transaction processing. It also provides a simple and concise view around particular subject issues by excluding data that are not useful in the decision support process.
- **Integrated:** Data warehouse is constructed by integrating multiple, heterogeneous data sources such as relational databases, flat files, on-line transaction records etc. Here data cleaning and data integration techniques are applied. Consistency in naming conventions, encoding structures, attribute measures etc. is also ensured among different data sources.
- **Time-variant:** The time horizon for the data warehouse is significantly longer than that of operational systems. Operational database store current value data, whereas warehouse data provide information from a historical perspective (e.g., past 5-10 years). Every key structure in the data warehouse contains an element of time, explicitly or implicitly, but the key of operational data may or may not contain "time element".
- **Nonvolatile:** A physically separate store of data is transformed from the operational environment to the data warehouse. Operational update of data does not occur in the data warehouse environment, because it does not require transaction processing, recovery, and concurrency control mechanisms. Rather it requires only *initial loading of data* and *access of data.*

2.1 Related Work

Research on database compression put a great role on performance improvement and storage reduction. A dictionary based compression technique HIBASE model [1], is further extended by the three-layer architecture [2] for sparsely populated and XML data. Migrating a multi terabyte archive from Object to relational database is depicted in [3]. A CMBVS architecture which is a disk based compression technique to handle terabyte level database is introduced in [4].

Alternatively, research on data warehouse distribution and parallelization is a modern issue in this time. Some of the distributed and parallel aspects of creating, maintaining, and querying a data warehouse are outlined in [5]. Teradata's parallel processing architecture [11] possesses distinct advantages for highly complex, large scale data warehouse. Cost optimization for distributed data warehouse is addressed in [7] which is based on a weighted sum of query and maintenance costs. WHIP architecture [12] performs identification, transformation and incremental integration of distributed heterogeneous data. SAND Technology [8] proposed new architecture of data warehouse that reduces storage requirements and therefore costs. Agent-Based Warehouse [10] architecture put forward a great contribution for seamless integration of new data sources.

Data cube computation algorithm mostly based on sorting and hashing techniques. PipeSort, Overlap and MemoryCube [14] are sort based where as Pipehash [11] is hashing based algorithm. Since sorting a terabyte sized data warehouse takes a huge computation time, our approach is based on hashing technique. Dwarf [15] and PrefixCube [16] separately present compact data cube structures and addresses the issues of prefix and suffix redundancy, but their structures store the lexemes of the dimension attributes which occupy massive space. On the contrary, in our approach, prefix and suffix redundancy is automatically abolished for the absence of lexemes. Recently, different fast parallel methods are developed for constructing data cubes [17, 18] based on partitioning and efficient parallel merge procedure with a cost of lexeme storage. This merge cost is much higher in case of very large scale data warehouse. In contrast, our approach is based on the vertical partitioning which completely avoid the redundancy of merging.

2.2 Scalability of Data Warehouse

In an environment where very large amounts of historical data need to be managed, Scalability is obviously a major concern. Today's warehouses are approaching several terabytes of data with a broader mix of query types, which is extreme as no-one has yet got close to requiring a petabyte (thousands of terabytes) sized data warehouse. Wal-Mart, for example, already has a data warehouse that contains hundreds of terabytes, yet it estimates that its implementation of RFID (radio frequency identification), once fully rolled out, will generate an additional 7Tb of data per day. While some of this additional data may be discarded, much of it will not, since it will be needed to analyze customer purchasing habits.

The impact of active data warehousing is not so much on the amount of data that needs to be stored but on the fact that it requires real-time updating of the warehouse. It means that the warehouse has to be able to support a broader mix of query types and, most pertinently, it means that there are far more users that need to access the warehouse. In other words, there is a major scalability issue, even if it is with respect to user numbers as opposed to data stored. The truth is that there are only a limited number of traditional approaches to the scalability/performance issues we have highlighted and we can categorize these as follows:

1. Split the problem up: One approach can best be categorized as breaking the problem into its constituent elements. Thus we might have a separate multi-dimensional database for aggregated queries, a separate analytic engine to cater for complex analytic queries, a separate operational data store, a separate store to support 'single customer view' type queries, a separate metadata repository, a separate master data system and so on. This approach will undoubtedly improve the performance of those queries for which there are specialized storage engines, which has the subsidiary advantage that it relieves strain from the main data warehouse. However, it does not really resolve the data storage issue and introduces new complexities that make the environment more difficult to manage, especially as it may mean storing data in more than one place.

2. Use federated data warehouses: the idea here is that we have several smaller data warehouses that talk to one another rather than one large one. The problem is that this is rarely appropriate because, for many organizations the whole purpose of a data warehouse is to gather information together so that it can be queried—it only makes sense if we can easily sub-divide the organization. Thus, it might make sense for a conglomerate that could sensibly treat its mining operations as wholly distinct from its cosmetics division, for example, but in most cases this will not apply.

3. Throw more hardware at it: This is the true traditional response to performance and scalability issues. The difference today is that vendors are talking about utility computing and grid computing. However, both of these approaches treat the symptoms rather than the disease. The disease is that data warehouses are too BIG and we need to find a way of reducing their size.

Therefore, our target should be: (i) to store data in a highly compressed form, typically taking up around 1/10th the space that would otherwise be required. (ii) to continue the query that data while it is still in the archive and without de-compressing it. (iii) to rapidly reconstitute the compressed data (or a subset thereof) back into the data warehouse (or into a new data mart) at any time to perform more complex queries on that data. To summarize, we would prefer to have data readily available in some fashion but, at the same time, we would also like to be able to dramatically reduce the size and cost of the storage needed for the warehouse.

2.3 Data Warehouse Architecture

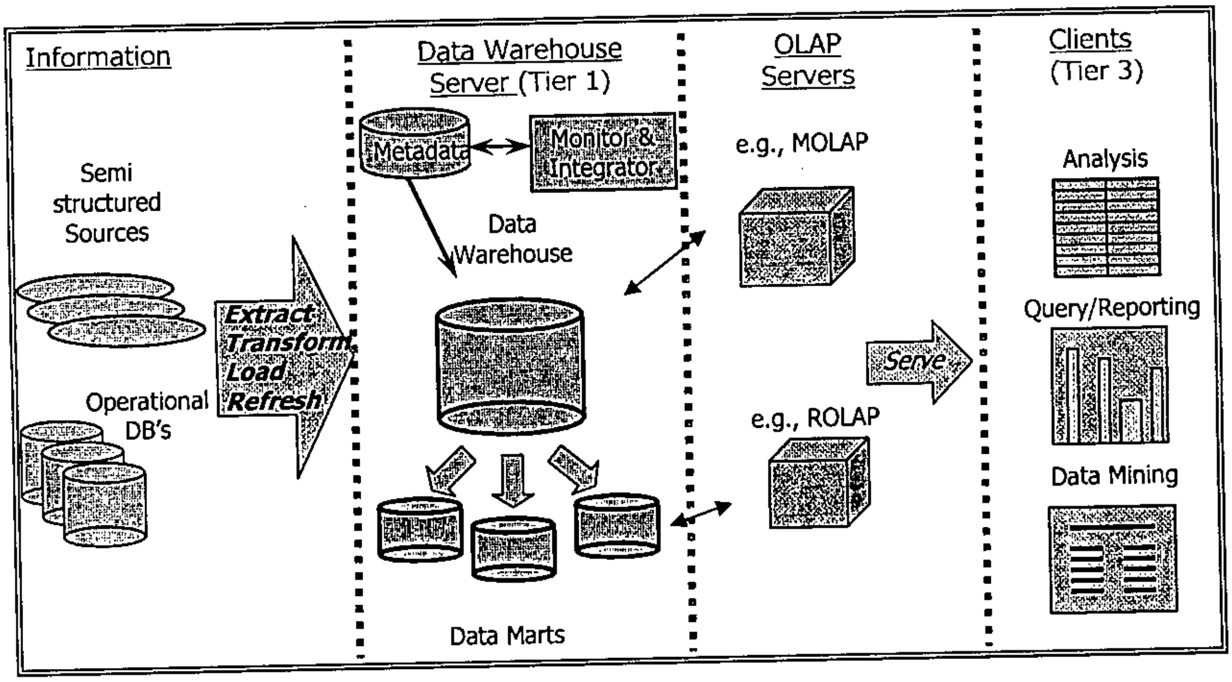


Fig. 2.1: General data warehouse architecture

The general data warehouse architecture is depicted in Fig. 2.1. Data is collected from external sources like semi structured and operational data, which are then extracted, transformed, loaded, refreshed and stored into the warehouse. The metadata is incorporated into the data warehouse through the monitoring and integration process. Data marts are generated by partitioning the warehouse data based on the date, business type, geography, etc. They are the subset of corporate-wide data that is of value to a specific group of users. Its scope is confined to specific, selected groups, such as marketing data mart. Later on, different OLAP (Online Analytical Processing) tools are generated, from which clients do their analysis, query processing and different data mining operations.

2.3.1 Noaman's Distributed Data Warehouse

Here the architecture provides a geographically distributed organization with a Distributed Detailed Data Warehouse (DDDW) and a Centralized Summarized Data Warehouse (CSDW) [6]. This is because in a geographically distributed organization, there are two types of decision makers: i) Those who make decisions for the entire organization, ii) Those who make decisions for one of the organization's branch.

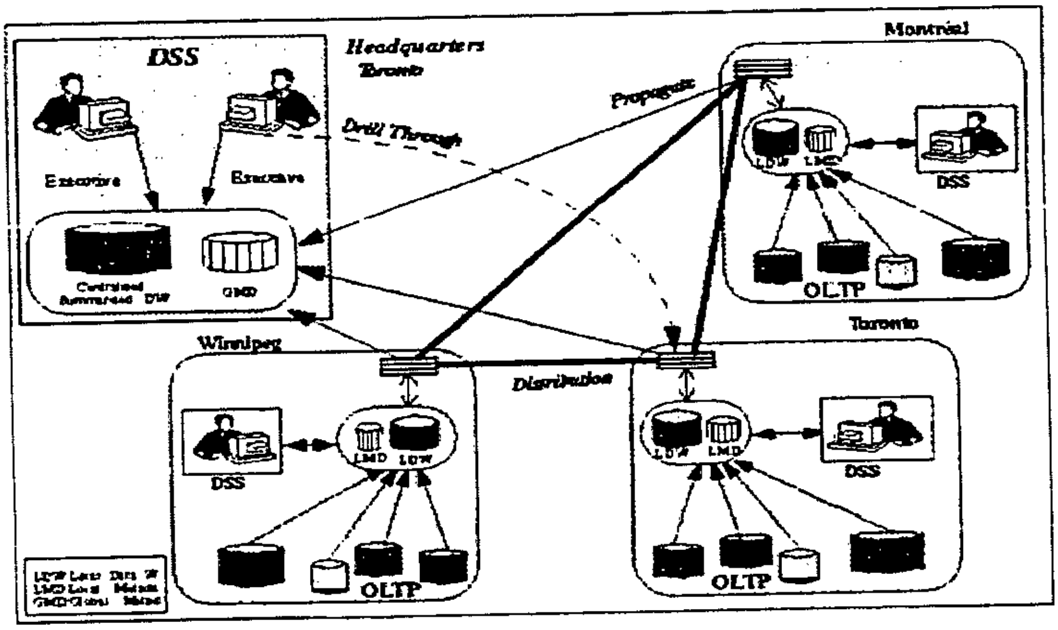


Fig. 2.2: Noaman's distributed data warehouse architecture [6].

In DDDW, organizations are connected by peer to peer approach. It supports two types of decision makers by providing single view of data (also called distribution transparency). In CSDW, summarized data (select/project/aggregated views) from the organization's branches are moved to headquarter where the CSDW is located. Hence, it provides quick response time, efficient analysis, and high security over the most sensitive data. Summarized data may throw away potentially useful information. Solution to this paradox is to link with DDDW that permits drill through.

2.3.2 Teradata's Massively Parallel Processing Architecture

Teradata [11] is an integrated solution which consists of distributed hardware platform and database software under either MP-RAS or Microsoft Windows server operating system. Teradata engineering team has spent the past 20 years extending and enhancing the scalability and performance of the Teradata system. The hardware platform uses Intel 32-bit processors configured in dual processor SMP (Symmetric Multiprocessing) nodes. Each node contains its own private memory, up to 4GB, and a private storage pool. Teradata systems scale from 1 to 512 nodes. The largest production Teradata system currently contains 296 nodes. The nodes communicate via a scalable high-speed communication interconnect called the BYNET.

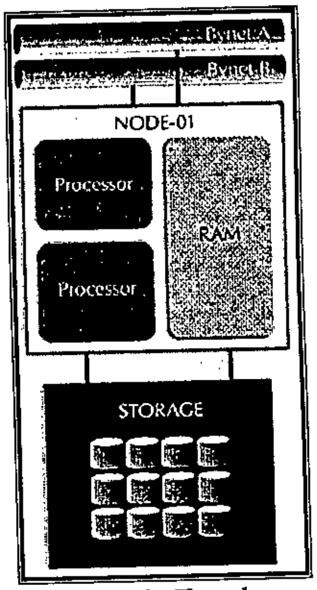


Fig. 2.3: Teradata system architecture [11]

2.3.3 Whips Warehouse Architecture

In Whips [12] warehouse architecture, the sources and warehouse views can be added and removed dynamically. It is scalable by adding more internal modules. The changes at the sources are detected automatically. Warehouse is always kept consistent with the source data by the integration algorithms. Therefore, the warehouse may be updated continuously. Each source is encapsulated by a source-specific monitor and wrapper. The monitor is used for detecting changes on the source data and notifying the integrator. The wrappers translate queries from the internal relational representation to the native language of its source. Views are defined through a view specifier. There is one view manager for maintaining each view. The integrator receives sources updates and forwards them to the relevant view managers. The query processor receives global queries from the view managers and answers back to the view managers by interacting with the source data through the wrapper. Therefore, it poses the appropriate single-source queries to the source wrappers, which are eventually answered back to the view managers.

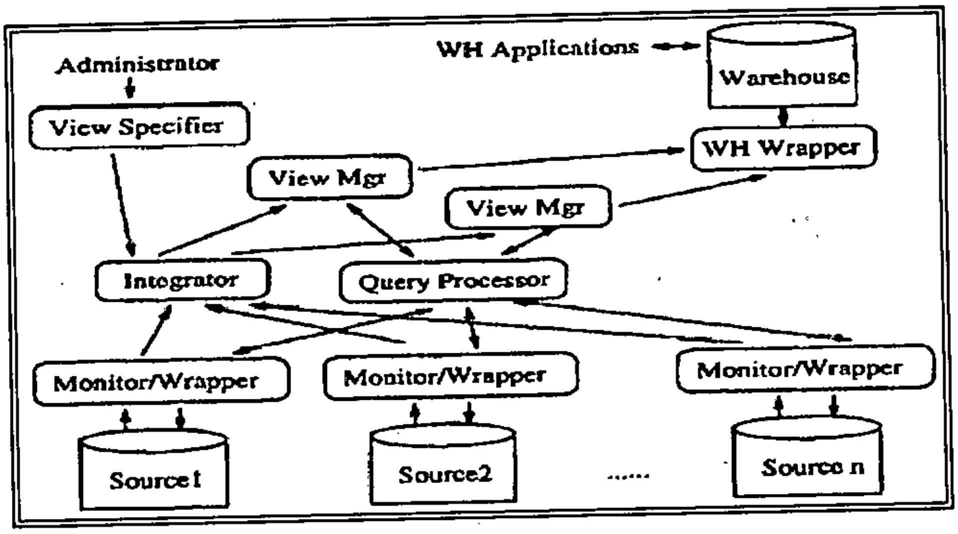


Fig. 2.4: Whips warehouse system architecture [12].

2.3.4 Object-Relational Data Warehouse

The differences of these architectures are "the object-orientation" approach and the new metadata layer. Objects of **Application Interface** layer receive user's queries, preprocess these queries and then send final request to the Data Warehouse Management component. Afterward, they obtain the queries results from the deeper layer.

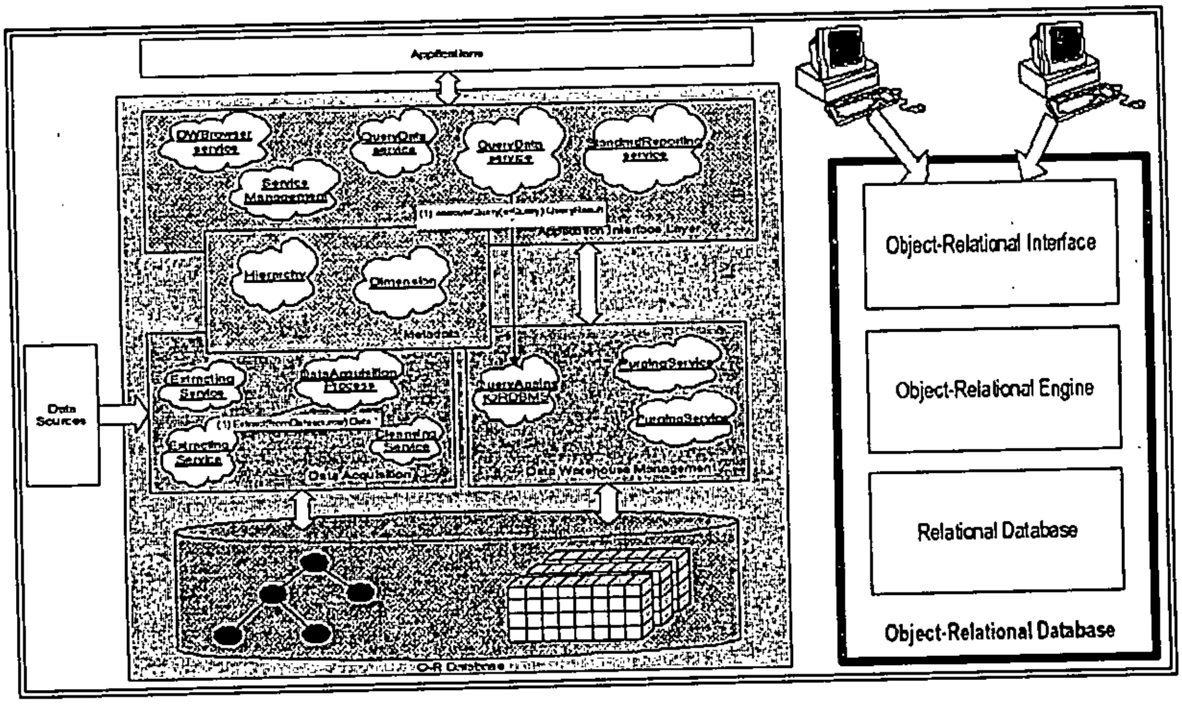


Fig. 2.5: Object-Relational data warehouse [3].

The **Data Acquisition** objects will extract, transform and load data from different legacy operational data stores (ODS) to the data warehouse O-R database. In **Data Warehouse Management** layer, the object directly accesses data from the O-R database. It provides services, which bridge the application interface layer and the O-R database. **Metadata** shows a mapping between object environment and relational environment. It is used by the query manager to generate an appropriate query. Various metadata classes are defined and discussed here. Technical metadata primarily supports technical staff that must implement and deploy the data warehouse. The business metadata primarily supports business end users who do not have a technical background, and cannot use the technical metadata to determine what information is stored inside the data warehouse. The information navigator metadata is a facility that allows users to browse through both the business metadata and the data inside the data warehouse. The data stored in **O-R DW** differs primarily from DW in relational environment and object-oriented data warehouse. Depending on the requirements and data types, O-R DW designers can decide to model it as a "cube", like MOLAP (Multidimensional OLAP), or as object hierarchy, like O3LAP (Object-Oriented OLAP).

2.3.5 Agent-Based Warehousing System

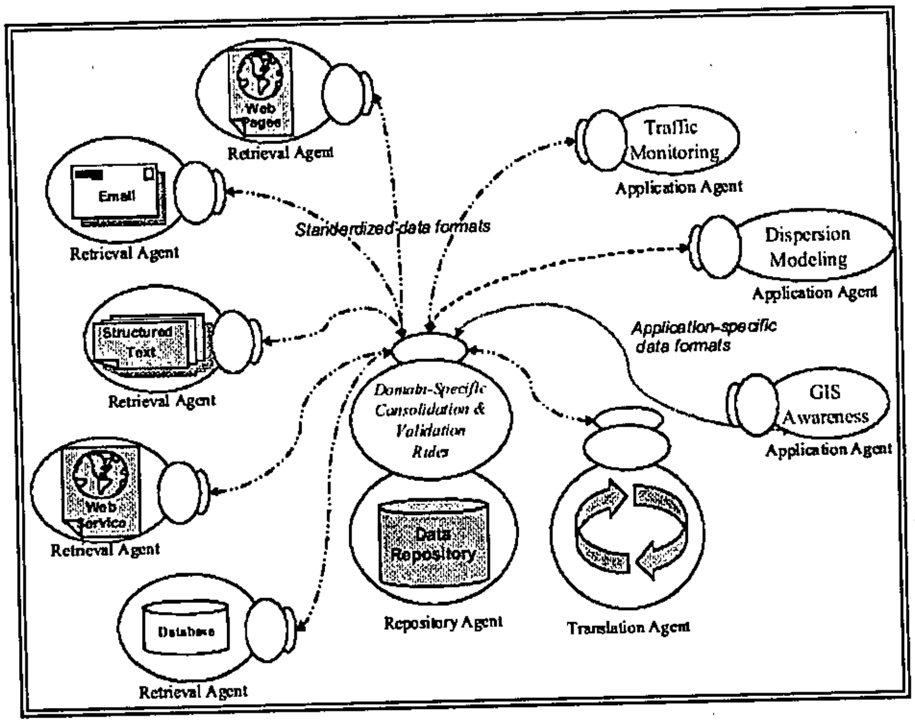


Fig. 2.6: Agent based data warehouse [10]

This system demonstrates how flexible agent architecture can be used for solving a number of problems associated with collecting data from a number of disparate, independently managed information sources, where the quality, format, and sources of the data may change over time. It allows for dynamically adding agents corresponding to new sources and data formats without altering any other system components. It allows sharing of data in different formats to a wide variety of applications. Retrieval agents provide data extraction, parsing and standardizing services. Repository agents maintain a large repository of data that they subscribe to in QLI Format. Translation agents provide data structure conversion services from QLI Format to other formats. Application agents are wrapper style representatives of various applications in the system.

2.4 Shared Nothing Architecture

There are three dominent themes in building high transaction rate multiprocessor systems, namely **shared memory** - multiple processors shared a common central memory (e.g. Synapse, IBM/AP configurations), **shared disk** - multiple processors each with private memory share a common collection of disks (e.g. VAX/cluster, any multi-ported disk system), and **shared nothing** - neither memory nor peripheral storage is shared among processors (e.g. Fault tolerant computer system). **Hierarchical model** is a hybrid of the preceding three architectures.

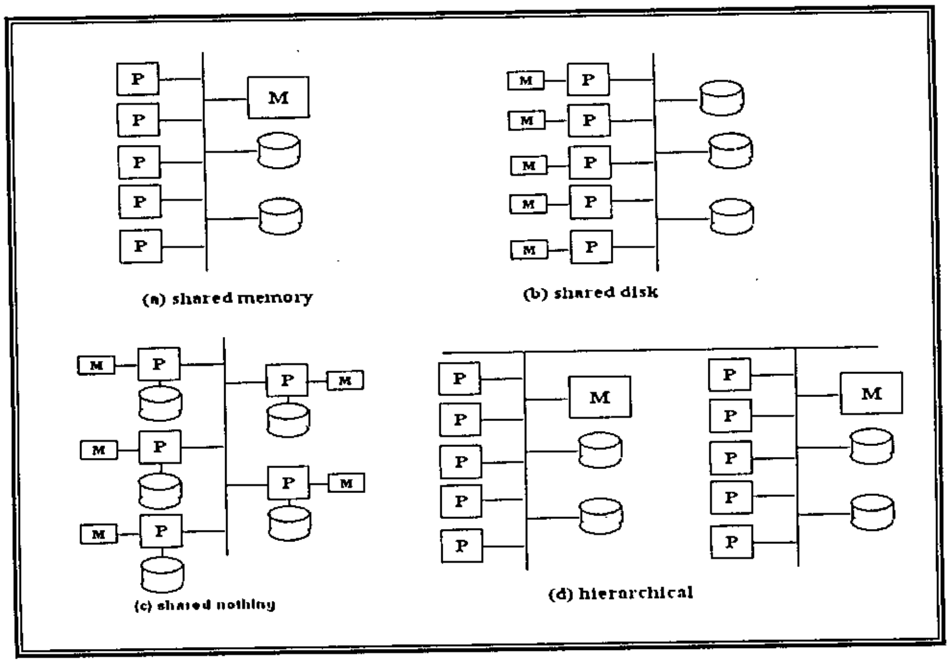


Fig. 2.7: Parallel database architectures

A **shared nothing architecture** (SN) is a distributed computing architecture where each node is independent and self-sufficient, and there is no single point of contention across the system. SN is popular for web development because of its scalability. As Google has demonstrated, a pure SN system can scale almost indefinitely simply by adding nodes in the form of inexpensive computers, since there's no single bottleneck to slow the system down. Two popular web development technologies, PHP and Ruby on Rails, both emphasize an SN approach, in contrast to technologies like J2EE that manage a lot of central state. An SN system may partition its data among many nodes (assigning different computers to deal with different users or queries), or may require every node to maintain its own copy of the application's data, using some kind of coordination protocol. In scalable, tunable, nearly delightful data bases, SN systems will have no apparent disadvantages compared to the other alternatives mentioned above. Hence the SN architecture adequately addresses the common case for the distributed data base system.

2.5 Data Warehouse Dimensions

A **dimension** is a structure that categorizes data in order to enable users to answer business questions. Commonly used dimensions are customers, products, and time. For example, each sales channel of a clothing retailer might gather and store data regarding sales and reclamations of their Cloth assortment. The retail chain management can build a data warehouse to analyze the sales of its products across all stores over time and help answer questions such as:

- What is the effect of promoting one product on the sale of a related product that is not promoted?
- What are the sales of a product before and after a promotion?
- How does a promotion affect the various distribution channels?

Generally the dimensional information itself is stored in a dimension table. In addition, the database object dimension helps to organize and group dimensional information into hierarchies. This represents natural `1:n` relationships between columns or column groups (the levels of a hierarchy) that cannot be represented with constraint conditions. Going up a level in the hierarchy is called rolling up the data and going down a level in the hierarchy is called drilling down the data. For example, Within the **`time`** **dimension**, months roll up to quarters, quarters roll up to years, and years roll up to all years. Within the `product` **dimension**, products roll up to subcategories, subcategories roll up to categories, and categories roll up to all products. Within the `customer` **dimension**, customers roll up to `city`, cities roll up to `state`, states roll up to `country`, countries roll up to `subregion`. Finally, subregions roll up to `region`, as shown in Fig. 2.8. Data analysis typically starts at higher levels in the dimensional hierarchy and gradually drills down if the situation warrants such analysis. Dimensions do not have to be defined. However, if our application uses dimensional modeling, it is worth spending time creating them as it can yield significant benefits, because they help query rewrite perform more complex types of rewrite.

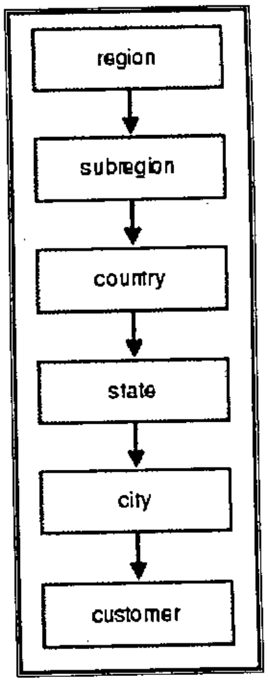


Fig. 2.8: Customer dimension

2.6 Data Warehouse Schemas

A schema is a collection of database objects, including tables, views, indexes, and synonyms. There is a variety of ways of arranging schema objects in the schema models designed for data warehousing which are stated below:

2.6.1 Third Normal Form Schema

Third normal form modeling is a classical relational-database modeling technique that minimizes data redundancy through normalization. When compared to a star schema, a 3NF schema typically has a larger number of tables due to this normalization process. For example, in Fig. 2.9, `orders` and `order items` tables contain similar information as `sales` table in the star schema in Fig. 2.10.

3NF schemas are typically chosen for large data warehouses, especially environments with significant data-loading requirements that are used to feed data marts and execute long-running queries. The main **advantages** of 3NF schemas are that they:

- Provide a neutral schema design, independent of any application or data-usage considerations
- May require less data-transformation than more normalized schemas such as star schemas

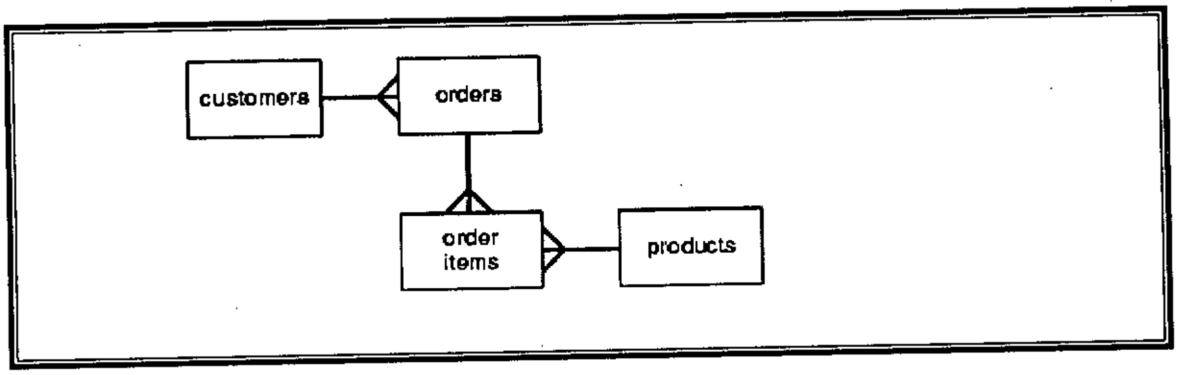


Fig. 2.9: Third Normal Form Schema

Queries on 3NF schemas are often very complex and involve a large number of tables. The performance of joins between large tables is thus a primary consideration when using 3NF schemas. One particularly important feature for 3NF schemas is partition-wise joins. The largest tables in a 3NF schema should be partitioned to enable partition-wise joins. The most common partitioning technique in these environments is composite range-hash partitioning for the largest tables, with the most-common join key chosen as the hash-partitioning key. Parallelism is often heavily utilized in 3NF environments, and parallelism should typically be enabled in these environments.

2.6.2 Star Schema

The **star schema** is perhaps the simplest data warehouse schema. It is called a star schema because the entity-relationship diagram of this schema resembles a star, with points radiating from a central table. The center of the star consists of a large fact table and the points of the star are the dimension tables. Star schemas are used for both simple data marts and very large data warehouses.

A **star query** is a join between a fact table and a number of dimension tables. Each dimension table is joined to the fact table using a primary key to foreign key join, but the dimension tables are not joined to each other. The optimizer recognizes star queries and generates efficient execution plans for them.

A typical **fact table** contains keys and measures. For example, in the sample schema in Fig. 10, the fact table, `sales`, contain the measures `quantity_sold`, `amount`, and `cost`, and the keys `cust_id`, `time_id`, `prod_id`, `channel_id`, and `promo_id`. The **dimension tables** are `customers`, `times`, `products`, `channels`, and `promotions`. The `products` dimension table, for example, contains information about each product number that appears in the fact table.

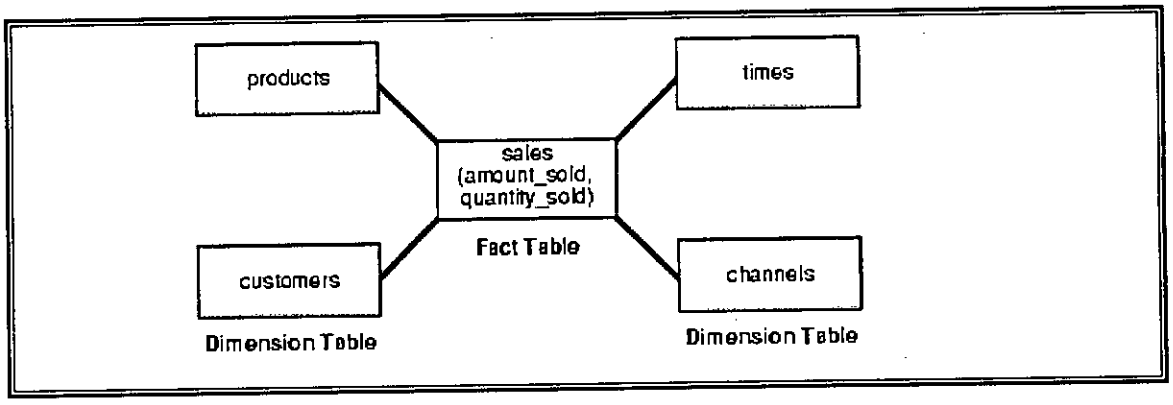


Fig. 2.10: Star Schema

A **star join** is a primary key to foreign key join of the dimension tables to a fact table. The main **advantages** of star schemas are that they:

- Provide a direct and intuitive mapping between the business entities being analyzed by end users and the schema design.
- Provide highly optimized performance for typical star queries.
- Easy to understand, easy to define hierarchies, reduces number of physical joins, low maintenance, very simple metadata.
- Are widely supported by a large number of business intelligence tools, which may anticipate or even require that the data warehouse schema contain dimension tables.

2.6.3 Snowflake Schema

The snowflake schema is a more complex data warehouse model than a star schema, and is a type of star schema. It is called a snowflake schema because the diagram of the schema resembles a snowflake. Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a products table, a product_category table, and a product_manufacturer table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The result is more complex queries and reduced query performance. Fig. 2.11 presents a graphical representation of a snowflake schema.

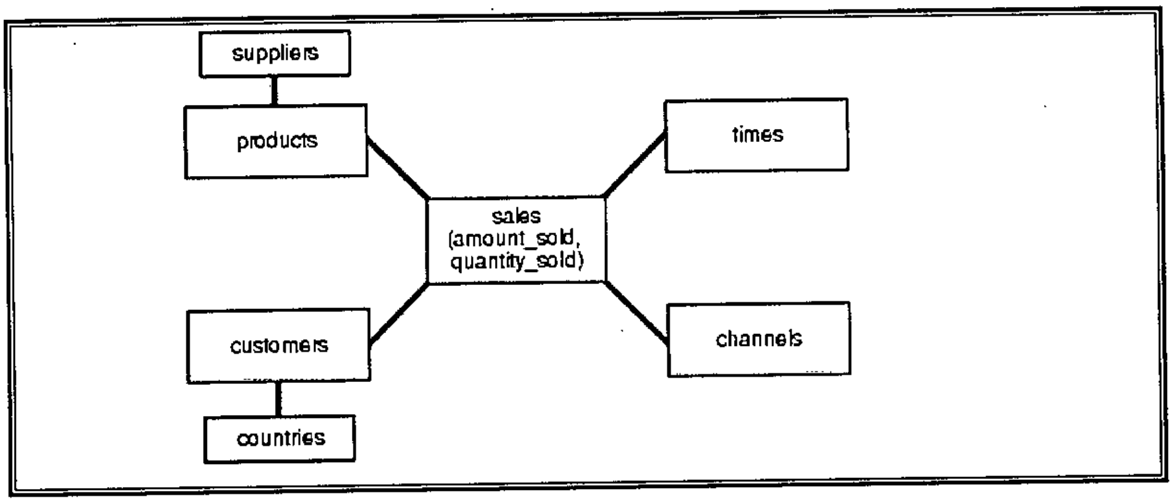


Fig. 2.11: Snowflake Schema

2.6.4 Fact Constellation Schema

Multiple fact tables share dimension tables, viewed as a collection of stars, therefore called galaxy schema or fact constellation. In the Fact Constellations, aggregate tables are created separately from the detail, therefore it is impossible to pick up, for example, Store detail when querying the District Fact Table according to Fig. 2.12. The major disadvantage is that dimension tables are very large in some cases, which can slow performance; front-end must be able to detect existence of aggregate facts, which requires more extensive metadata.

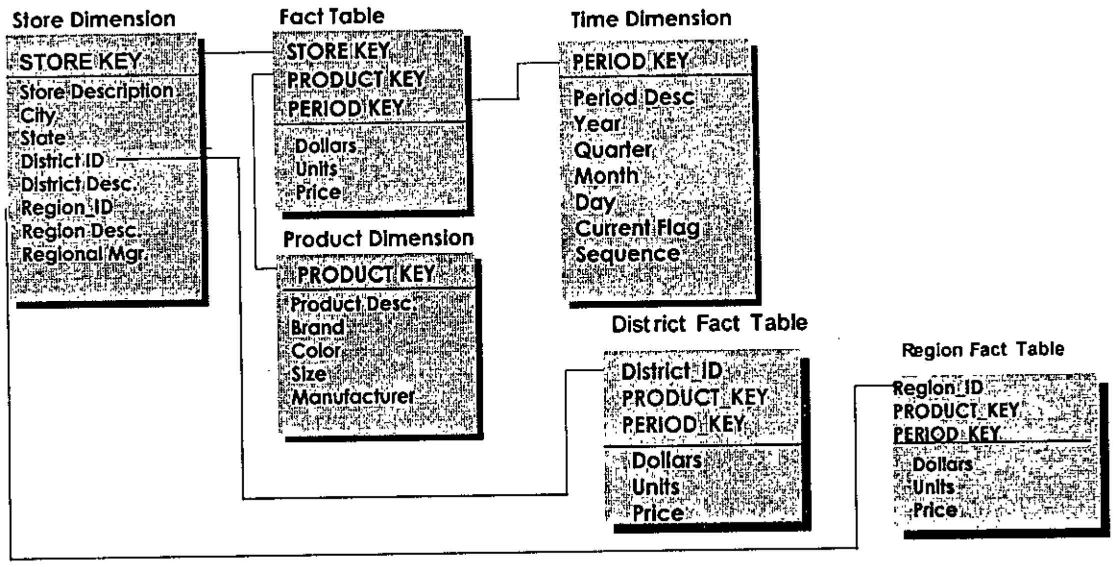


Fig. 2.12: Fact Constellation Schema

2.7 Multidimensional Data Cube

A data warehouse is based on a multidimensional data model which views data in the form of a data cube. A data cube allows data to be modeled and viewed in multiple dimensions. The CUBE BY is a multidimensional extension of the relational operator GROUP BY, which computes GROUP BY on all possible combinations of grouping attributes in the CUBE BY clause. Each group-by is called a cuboid. In data warehousing literature, an n-D base cube is called a base cuboid. The top most 0-D cuboid, which holds the highest-level of summarization, is called the apex cuboid. The lattice of cuboids forms a data cube.

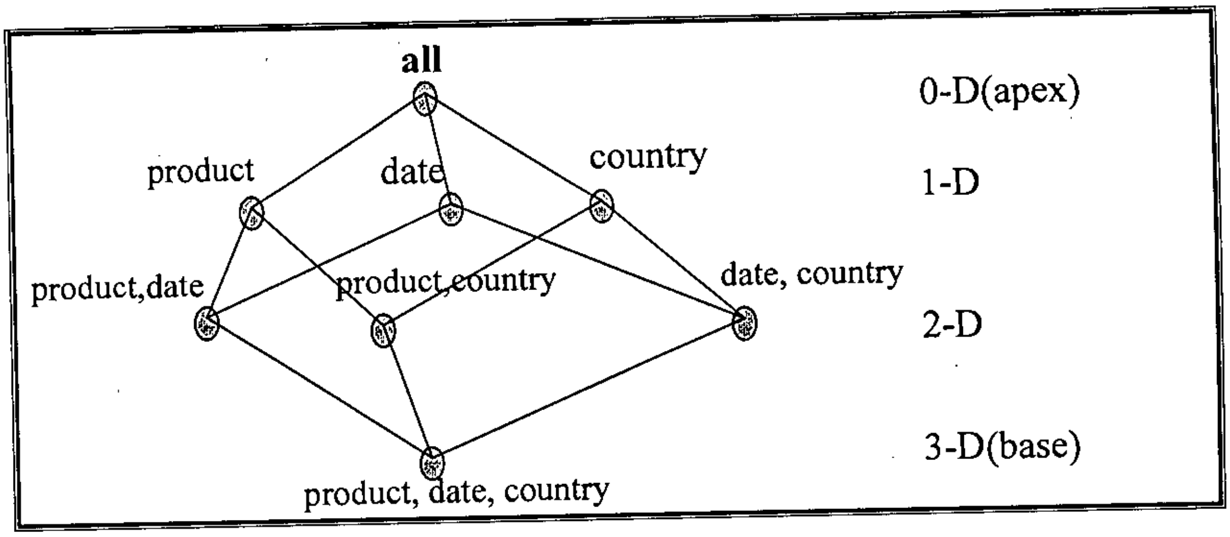


Fig. 2.13: Cuboids corresponding to the data cube

The CUBE BY is a very expensive operator, and the size of its result is extremely large especially when the number of CUBE BY attributes and the number of tuples in the source relation is large. Therefore, we need to materialize the data cube at least partially. But selection of which cuboids to materialize is based on size, sharing, access frequency, etc. In data warehousing literature, an n-D base cube is called a base cuboid.

2.8 Online Analytical Processing (OLAP)

OLAP provides advanced data analysis environment. It supports decision making, business modeling, and operations research activities. The characteristics of OLAP are:

- Use multidimensional data analysis techniques
- Provide advanced database support

- Provide easy-to-use end-user interfaces
- Support complex, ad hoc queries.
    - Access large amounts of data (5 years of sales data).
    - Analyze relation between many types of business elements (sales, products, region, and channels).
    - Compute and compare aggregated data over hierarchical time periods (monthly, quarterly, yearly)
    - Present data in various perspectives (sales by region vs. sales by channels within each region).

The typical OLAP operations are:

- Roll up (drill-up): Summarize data by climbing up hierarchy or by dimension reduction.
- Drill down (roll down): Reverse of roll-up. Here from higher level summary to lower level summary or detailed data are achieved.
- Slice and dice: Here project and select operations are performed.
- Pivot (rotate): Reorient the cube, visualization, 3D to series of 2D planes.
- drill across: involving (across) more than one fact table
- drill through: through the bottom level of the cube to its back-end relational tables (using SQL)

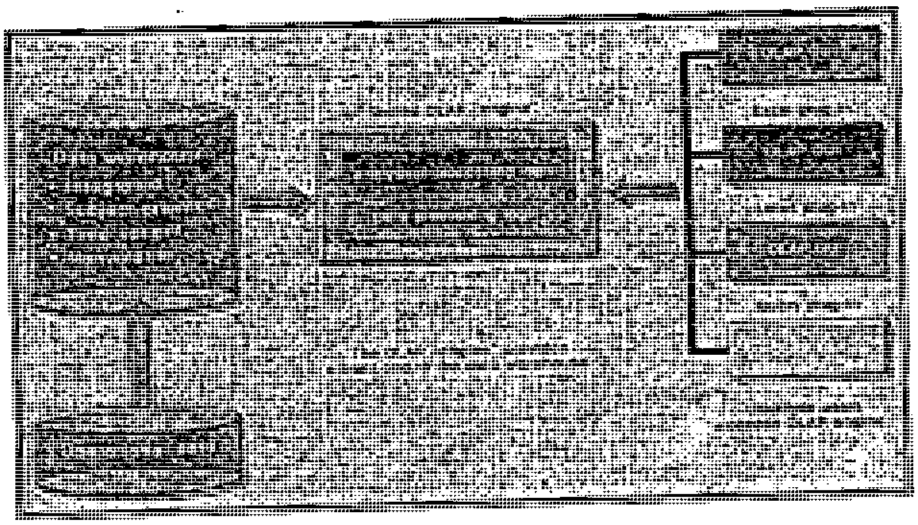


Fig. 2.14: OLAP Server arrangement.

2.8.1 Multidimensional OLAP (MOLAP)

Since data cubes are such a useful interpretation tool, most OLAP products are built around a structure in which the cube is modeled as a multidimensional array. These multidimensional OLAP, or MOLAP, products typically run faster than other approaches, primarily because it's possible to index directly into the data cube's structure to collect subsets of data.

However, for very large data sets with many dimensions, MOLAP solutions aren't always so effective. As the number of dimensions increases, the cube becomes sparser—that is, many cells representing specific attribute combinations are empty, containing no aggregated data. As with other types of sparse databases, this tends to increase storage requirements, sometimes to unacceptable levels. Compression techniques can help, but using them tends to destroy MOLAP's natural indexing

2.8.2 Relational OLAP (ROLAP)

Data cubes can be built in other ways. Relational OLAP uses the relational database model. The ROLAP data cube is implemented as a collection of relational tables (up to twice as many as the number of dimensions) instead of as a multidimensional array. Each of these tables, called a cuboid, represents a particular view.

Because the cuboids are conventional database tables, we can process and query them using traditional RDBMS techniques, such as indexes and joins. This format is likely to be efficient for large data collections, since the tables must include only data cube cells that actually contain data.

However, ROLAP cubes lack the built-in indexing of a MOLAP implementation. Instead, each record in a given table must contain all attribute values in addition to any aggregated or summary values. This extra overhead may offset some of the space savings, and the absence of an implicit index means that we must provide one explicitly.

2.9 Data Warehouse Compression

The Tera-scale Scientific Data Management [19] uses the compression to archive the data files in web servers, the query cannot be processed directly in the compressed files. The decompression is required to make any searching on the data.

Oracle uses block level dictionary based compression technique. The compression model incurs some redundancy but a block itself contains all information that is required to decompress. The compression of Oracle database is to manage large data sets. The details of the compression model are discussed in section 2.9.5.

The Kx Database Technology uses columnar format [20] to store large database but does not apply any compression mechanism. Though the columnar format really exhibits high performance in data access, it is not as fast as the compressed columnar format shows.

The architecture for multi terabyte, hierarchical data warehouse for continuous, high-rate object stream archiving relational data in a hierarchical storage system composed of serialized objects that have been binned and indexed is described in [21]. The architecture does not apply any compression technique, so obviously the I/O access rate is high and thus reduces the query performance.

A column oriented compression method C-Store [22] improves the performance of the DBMS. It is designed mainly to support high performance query processing but has no guideline about terabyte data management.

2.9.1 Row Store vs Column Store Compression

Database can be compressed in two ways: row or column wise. The benefits of row store compression are:

- It is Write Optimized.
- We can Insert/delete a record in one physical write.
- Good for OLTP

But it is not good for read most applications like: Data warehouses, CRM (Customer Relationship Management). However it can be improved by Bitmap indices, better sequential read, integration of "data cube" products, materialized views etc. DB2, Oracle, Sybase, SQLServer etc. use the Row Store compression technique. Nevertheless, there may be a better idea and that is Column Store compression technique. Since, Column Stores are Read Optimized. Ad-hoc queries read about 10% of the total number of columns. So, Column store reads 10% of what a row store reads.

2.9.2 The Implication of Compression on Database Processing

Compression has become now an essential part of many large information systems where large amount of data need to be processed stored or transferred. The data may be of any type e.g. voice, video, text, XML, table, etc. No single compression technique is suitable for all types of data. Lossy compression is use for voice or video data whereas loss-less compression is for most other data types.

According to Shannon [23], it has been known that the amount of information is not synonymous with the volume of data. Reducing the volume of data without losing any information is known as loss-less data compression. Loss-less data compression can be attractive for two reasons: data storage reduction and performance improvements. Storage reduction is a direct and obvious benefit but performance improvement arise as follows:

- Main memory access time is in the range of nanoseconds while disk access is in the range milliseconds. Only a smaller amount of compressed data needs to be transferred and/or processed to effect any particular operation. Thus, it is improving I/O processing and hence improving the overall performance.
- A further significant factor is now arising in distributed application using mobile and wireless communication. Low bandwidth is a performance bottleneck and data transfers may be costly. Both of these factors make data compression well worthwhile.

Combining compression with data processing improves performance. Database systems require providing efficient addressability for data, and generally must provide dynamic update. It has proved difficult to combine these with good compression technologies. Much research work has been done in database systems to exploit the benefit of compression in storage reduction and performance improvement [2].

2.9.3 Compression of Relational Structure

Relational structure can be benefited using compression as follows:

Significant improvement occurs in index structures such as B-tree and R-tree by reducing the number of leaf pages. Reduction in transaction turnaround time and user response time as a result of faster transfer between disk and main memory in I/O bound system. In addition, since this will reduce I/O channel loading, the CPU can process many more I/O systems and thus channel utilization is increased [24].

Improvement in the efficiency of backup since copies of the database could be kept in compressed form. This reduces the number of tapes required to store the data and reduces the time of reading from, and writing to, these tapes [25].

Processing data in compressed form makes it possible for the whole, or the major part of a database to be memory resident. Main memory access time several orders of magnitudes faster than the secondary storage access time. Thus improves performance.

The horizontal organization has the advantage that a single disk access fetches all attribute values in a tuple. Even for non-compressed memory resident databases, where memory access speeds are much faster, the implementation simplicity is appreciable. A relation can also be represented as a sequence of column vectors in which corresponding attribute values in successive tuples is stored adjacently [1].

In a main memory database system (MMDB) the data resides permanently in main physical memory and memory resident data may have a back up copy on the disk. In conventional disk based database systems the primary copy of the data resides in the disk and a small portion of data is memory resident. The distinguishing features of MMDB described in [26] are the access time of main memory is several orders of magnitude less than for disk storage. Disk has a high fixed cost per access that does not depend on the amount of data that is retrieved during the access.

2.9.4 Table Data Compression

Effective exploratory analysis of massive, high-dimensional tables of alpha-numeric data is a ubiquitous requirement for a variety of application environments for example of massive data tables is Call Detail Records of large telecommunication systems. A typical CDR is a fixed length record structure comprising several hundred bytes of data that capture information on various attributes of each call. These include network level information (e.g., end point exchanges), time-stamp information and billing information. These CDRs are stored in tables that can grow to truly massive sizes, in the order of several terabytes per year. Database compression differs from table compression in many ways.

Database compression stresses the preservation of indexing-the ability to retrieve an arbitrary record under compression. Table compression does not require indexing to be preserved. Database records are often dynamic but the table data are write-once discipline. Databases consist of heterogeneous data whereas table data are more homogeneous, with fixed field lengths. Unlike table data, databases are not routinely terabytes in size. The approach to database compression requires either lightweight techniques such as compressing each tuple by simple encoding or compression of the entire table. These approaches are not appropriate for table compression: the former is too wasteful and the latter is too expensive.

Table compression was introduced by Buchsbaum et.al. [27], as a unique application of compression based on several distinguishing characteristics. They have introduced a system called pzip. The basis of the method is to construct a compression plan studying a very small training set off-line. The plan is based on data dependency. They have defined two types of data dependency: combinational and differential. If two data intervals have the compressed size in separate intervals larger than the compressed size in a combined single interval, the intervals are combinational dependent. Differential dependency is an explicit dependency between columns. Though pzip outperforms gzip by a factor of two, partitioning the data determining the dependency is a problem. Optimum partitioning is not possible using the training data set. The primary focus is to optimize the compression ratio within a user defined error bound. However none of these methods can be applied to databases where no loss of information is permissible.

Sparsely-populated table data arises when sometimes data is represented using a single horizontal table. An example is the sparse bit-map for a digital library, the electronic marketplace in and the news-portal system in IBM-Almaden [28]. Sparsely populated data can be compressed using run-length encoding method given in [1].

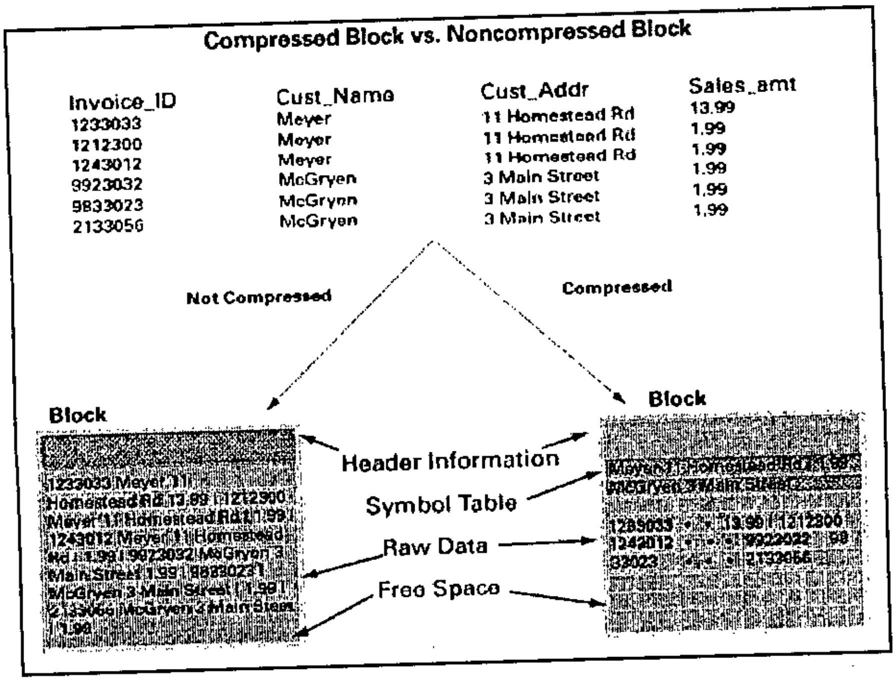
2.9.5 Compression in Oracle Database

The Oracle RDBMS recently introduced an innovative compression technique for reducing the size of relational tables [29]. By using a compression algorithm specifically designed for relational data, Oracle is able to compress data much more effectively than standard compression techniques. More significantly, unlike other compression techniques, Oracle incurs virtually no performance penalty for SQL queries accessing compressed tables. In fact, Oracle's compression may provide performance gains for queries accessing large amounts of data, as well as for certain data management operations like backup and recovery.

In past commercial database systems have not heavily utilized compression techniques on data stored in relational tables. A standard compression technique may offer space savings, but only at a cost of much increased query elapsed time. Hence, this trade-off has made compression not always attractive for relational databases.

Meikel Poess and Dmitry Patapov et.al. [29] recently describe how to compress Oracle table data in [29], that is an attractive solution for large relational data warehouses. It can be used to compress tables, table partitions and materialized view.

The compression algorithm used in Oracle for large data warehouse tables compresses data by eliminating duplicate values in database block (or page). The algorithm is lossless dictionary-based compression technique. The compression window for which a dictionary (symbol table) is created consists of one database block. Therefore, compressed data stored in a database block is self contained. That is, all the information needed to recreate the uncompressed data in a block is available within that block.



Invoice_ID	Cust_Name	Cust_Addr	Sales_amt
1233033	Meyer	11 Homestead Rd	13.99
1212300	Meyer	11 Homestead Rd	1.99
1243012	Meyer	11 Homestead Rd	1.99
9923032	McGryen	3 Main Street	1.99
9833023	McGryen	3 Main Street	1.99
2133056	McGryen	3 Main Street	1.99

Fig. 2.15: Oracle Database Compression.

Fig. 2.15 illustrates the differences between compressed versus non-compressed block. The top part of the Fig. 2.15 shows a typical data warehouse like fact table with row

ID, invoice Id, customer first name, customer last name and sales amount. There are entries for five customers showing six purchases. For data warehouse fact table it is very common to have this highly denormalized structure.

The bottom right part shows how the same data is stored in a compressed block. Instead of storing all data, redundant information is replaced by links to a common reference in the symbol table, indicated by the black dots. For each column value in all columns, based on length and number of occurrences in one block, the algorithm decides whether to create an entry into the symbol table for this column value. If column values from different columns have the same values, they share the same symbol table entry. This is referred to as cross-column compression. Only entire column values or sequences are compressed. Sequences of columns are compressed as one entity if a sequence of column values occurs multiple times in many rows. This is referred to as multi-column compression. This optimization is particularly beneficial for OLAP type materialized views using grouping sets and cube operators. For instance a cube of a table often repeats the same values along dimensions creating many potential multi-column values. Multi-column compression can significantly increase the compression factor and query performance. In order to increase multi-column compression, columns might be reordered within one block. For short column values and those with few occurrences no symbol table entry is created limiting the overhead of the symbol table and ensuring that compressing a table never increases its size. However, this is transparent to any application. This method improves I/O performance but the query processing requires a decompression on page by page basis.

2.9.6 Dictionary based HIBASE Compression Approach

The HIBASE [1] approach is a more radical attempt to model the data representation that is supported by information theory. The architecture represents a relation table in storage as a set of columns, not a set of rows. Of course, the user is free to regard the table as a set of rows. However, the operation of the database can be made considerably more efficient when the storage allocation is by columns.

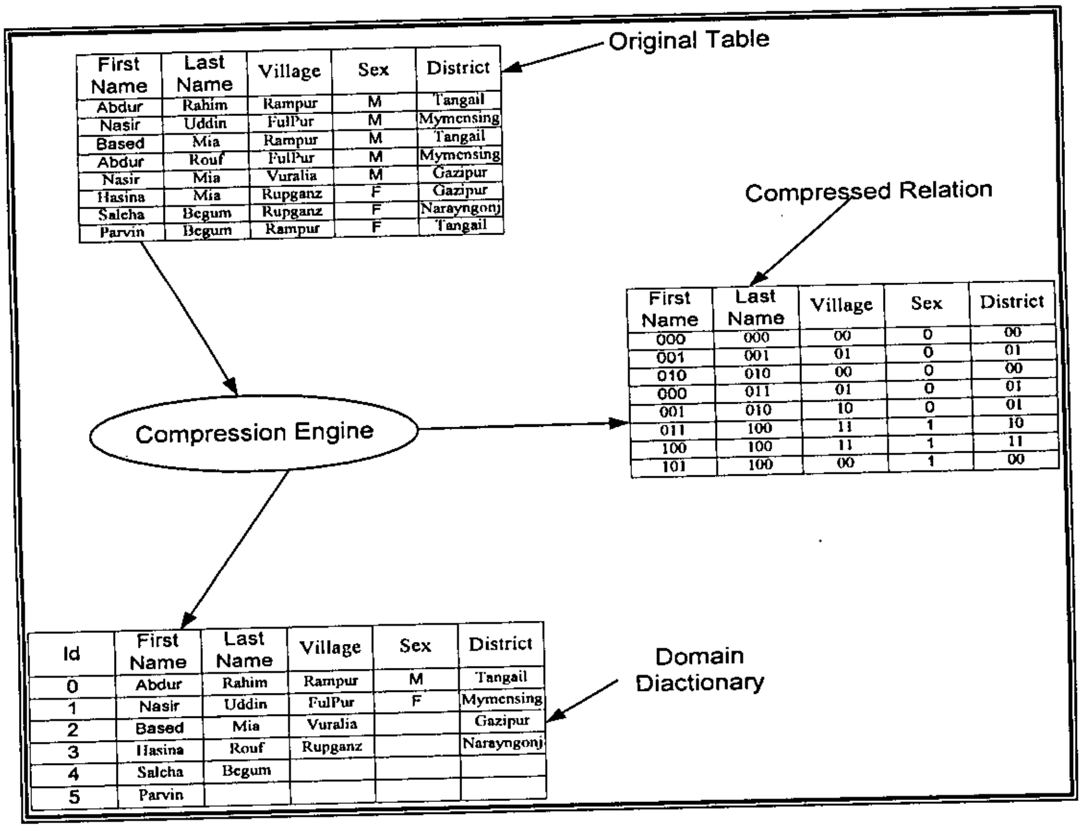
The database is a set of relations. A relation is a set of tuples. A tuple in a relation represents a relationship among a set of values. The corresponding values of each tuple belong to a domain for which there is a set of permitted values. If the domains are D_1,

D_2,……., D_n respectively. A relation r is defined as a subset of the Cartesian product of the domains. Thus r is defined as $r \subseteq D_1 \times D_2 \times \ldots\ldots\times D_n$.

2.9.6.1 HIBASE Compression Architecture

The architecture given in Fig. 2.16 is a compact representation that can be derived from a conventional record structure using the following steps:

- A dictionary per domain is employed to store the string values and to provide integer identifiers for them. This achieves a lower range of identifier, and hence a more compact representation than could be achieved if a single dictionary was provided for the whole database.
- Replace the original field value of the relation by identifiers. The range of the identifiers is sufficient to distinguish string of the domain dictionary. The example given in Fig. 2.16 shows that there are only seven distinct first names, so only a 3-bit can represent this attribute.



First Name	Last Name	Village	Sex	District
Abdur	Rahim	Rampur	M	Tangail
Nasir	Uddin	FulPur	M	Mymensing
Based	Mia	Rampur	M	Tangail
Abdur	Rouf	FulPur	M	Mymensing
Nasir	Mia	Vuralia	M	Gazipur
Hasina	Mia	Rupganz	F	Gazipur
Salcha	Begum	Rupganz	F	Narayngonj
Parvin	Begum	Rampur	F	Tangail

First Name	Last Name	Village	Sex	District
000	000	00	0	00
001	001	01	0	01
010	010	00	0	00
000	011	01	0	01
001	010	10	0	01
011	100	11	1	10
100	100	11	1	11
101	100	00	1	00

Id	First Name	Last Name	Village	Sex	District
0	Abdur	Rahim	Rampur	M	Tangail
1	Nasir	Uddin	FulPur	F	Mymensing
2	Based	Mia	Vuralia		Gazipur
3	Hasina	Rouf	Rupganz		Naraygonj
4	Salcha	Begum			
5	Parvin				

Fig. 2.16: Compression of a Relation Using Domain Dictionaries.

Hence in the compressed table each tuple resume only 3 bits for First Name, 3 bits for last Name, 2 bits for village, 2 bit for district and 1 bit for sex forming total of 11 bits. This is not the overall storage; however, we must take account of the space occupied by the domain dictionaries and indexes. Typically, a proportion of domain is present in several relations and this reduces the dictionary overhead by sharing it by different attributes.

2.9.6.2 Dictionary Structure in HIBASE

HIBASE used two alternative representation of the values stored in attribute: token and lexemes. A token is a sub-range of integers represented in its minimal binary encoding. Lexemes are a sequence of 0 or more 8-bit characters. The translation of string value to token is optimized using the minimal number of stored bits. String of decimal digit can be directly converted into binary at the database designer's discretion. Other data, which can not be represented directly as an integer, such as character string or real, are translated using dictionaries although in principle, real can be encoded as primitive data type.

A Dictionary must have Three Characteristics:

- It should map attribute value to their encoded representation during the compression operation: encode (lexeme)→token.
- It should perform the reverse mapping from codes to literal values when parts of the relation are printed out. Decode (token)→lexeme.
- The mapping must be cyclic such that x=encode (decode (x)).

The structure is attractive for low cardinality data. For high cardinality and primary key data, the size of the string heap grows considerably and contributes very little or no compression. The HIBASE compression does not support Unicode which is essential for current database applications.

2.9.7 Terabyte Data Management System Using CMBVS

In Columnar Multi-block Vector Structure (CMBVS), the multi-block vector structure stores the relation in column wise format. Each attribute is associated to a domain dictionary and a dynamic vector. The vector is partitioned into multiple blocks. The vector accommodates a certain number of blocks in memory. When the number of blocks are increased, some blocks are swapped to disk. The high cardinality of domain dictionaries are also partitioned into blocks if necessary. The original tuple value is first searched in the dictionary, if the value is found in dictionary corresponding token is returned. If the value is not is not found in the dictionary, it will be inserted into the dictionary and corresponding token will be returned. It will then be inserted into the vector. The detailed CMBVS architecture is plotted in Fig. 2.17.

The query can be carried out directly on the compressed form of data. The query may be optimized if there are multiple attributes are accessed in the query. The query in the CMBVS only accesses the associated vector and dictionary, so that the I/O access is minimized and hence increased the query throughput.

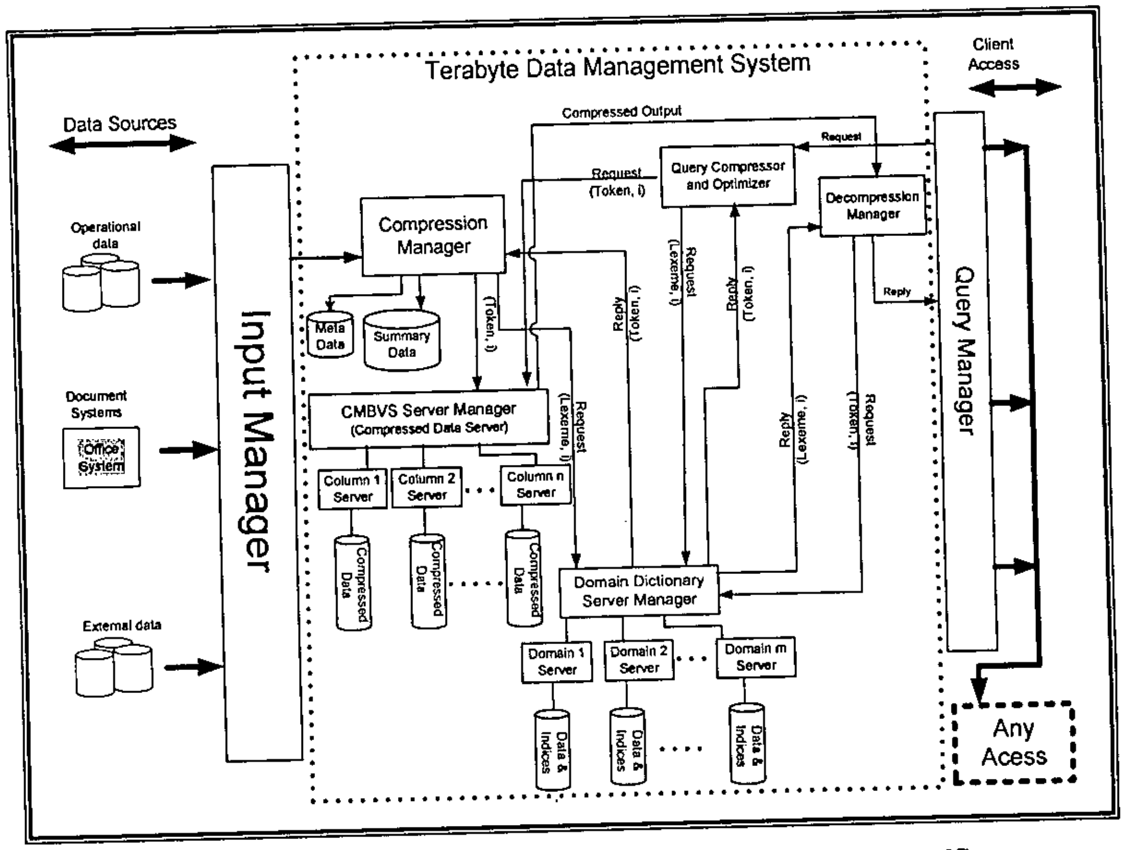


Fig. 2.17: Terabyte Data Management System using CMBVS

The CMBVS is scalable to very large number of tuples in compressed form. Even a microcomputer (with 120 GB hard disk) can host huge data sets in compressed form like terabyte in size. However, the performance degrades as the database size increases.

The CMBVS provides the infrastructure that can be used in distributed system to manage multi-terabyte database. As the relations are stored in columnar format, the multi-block vectors can be distributed over column severs. The domain dictionaries are distributed over multiple domain servers. The system is scalable to virtually unlimited number of tuples, by increasing the number of disks in disk array of both column servers and domain servers.

2.10 Parallel Computation

Parallel systems improve processing and I/O speeds by using multiple CPUs and disks in parallel. Parallel machines are becoming increasingly common, making the study of parallel database systems correspondingly more important. The driving force behind parallel database systems is the demands of applications that have to process an extremely large databases or that have to process an extremely large number of transactions per second (of the order of thousands of transactions per second). Centralized and client- server database systems are not powerful enough to handle such applications.

Distributed databases bring the advantage of distributed computing to the database management domain. A distributed computing system consists of a number of processing elements, not necessarily homogeneous, that are interconnected by a computer network, and that cooperate in performing certain assigned tasks. As a general goal, distributed computing systems partitions a big, unmanageable problem into smaller pieces and solve it efficiently in a coordinated manner. The economic viability of this approach stems from two reasons: (i) more computer power is harnessed to solve a complex task, and (ii) each autonomous processing element can be managed independently and develop its own applications. We can define a distributed database (DDB) as a collection of multiple logically interrelated databases distributed over a computer network, and a distributed database while making the distribution transparent to the user

2.10.1 Network programming

The **Internet Protocol (IP)** is at the core of network programming. IP is the vehicle that transports data between systems, whether within a local area network (LAN) environment or a wide area network (WAN) environment. Though there are other network protocols available to the Windows network programmer, IP provides the most robust technique for sending data between network devices, especially if they are located across the Internet. Programming using IP is often a complicated process. There are many factors to consider concerning how data is sent on the network: the number of client and server devices, the type of network, network congestion, and error conditions on the network. Because all these elements affect the transportation of data from one device to another, understanding their impact is crucial to success in network programming.

A **socket** is a connection between two hosts. It can perform seven basic operations: Connection to remote machine, Send data, Receive data, Close a connection, Bind to a port, Listen for incoming data, Accept connections from remote machines on the bound port. Each socket is associated with a particular protocol either UDP or TCP.

The most important thing to remember about using TCP is that it is a connection-oriented protocol. Once a connection exists between two devices, a reliable data stream is established to ensure that data is accurately moved from one device to the other. Although with TCP our applications do not need to worry about lost or out-of-order data, there is one huge consideration when we are programming with TCP: the buffers.

Because TCP must ensure the integrity of the data, it keeps all sent data in a local buffer until a positive acknowledgement of reception is received from the remote device. Similarly, when receiving data from the network, TCP must keep a local buffer of received data to ensure that all of the pieces are received in order before passing the data to the application program.

2.10.2 Threading and Multitasking

A thread is a series of instructions executed independently of any other instructions within an application. An application always has at least one thread of execution, but may have many more. Multithreading allows an application to divide tasks such that they work independently of each other to make the most efficient use of the processor and the user's time.

Multitasking is the simultaneous execution of multiple threads. Multitasking comes in two flavors: cooperative and preemptive. Very early versions of Microsoft Windows supported cooperative multitasking, which meant that each thread was responsible for relinquishing control to the processor so that it could process other threads. However, Microsoft Windows NT and later - Windows 98, Windows 2000, and Windows XP support the same preemptive multitasking that OS/2 does. With preemptive multitasking, the processor is responsible for giving each thread a certain amount of time (a time slice) in which to execute. The processor then switches among the different threads, giving each its time slice, and the programmer doesn't have to worry about how and when to relinquish control so that other threads can run. JAVA only work on preemptive multitasking operating systems, this is what we are focusing on.

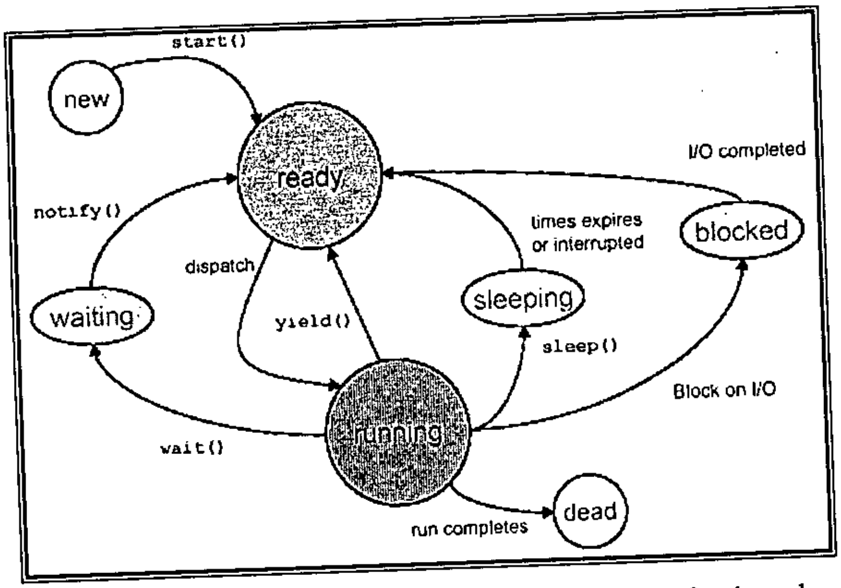


Fig. 2.18: State diagram showing the Life cycle of a thread.

Life Cycle of a Thread: At any time, a thread is said to be in one of several thread states (illustrated in Fig. 2.18). Let us say that a thread that was just created is in the born state. The thread remains in this state until the program calls the thread's start method, which causes the thread to enter the ready state (also known as the run able state). The highest priority ready thread enters the running state (i.e., the thread begins executing), when the system assigns a processor to the thread. A thread enters the dead state when its run method completes or terminates for any reason—a dead thread eventually will be disposed of by the system. One common way for a running thread to enter the blocked state is when the thread issues an input/output request. In this case, a blocked thread becomes ready when the I/O for which it is waiting completes. A blocked thread cannot use a processor even if one is available. When a running thread calls wait, the thread enters a waiting state for the particular object on which wait was called. One thread in the waiting state for a particular object becomes ready on a call to notify issued by another thread associated with that object. Every thread in the waiting state for a given object becomes ready on a call to notify all by another thread associated with that object.

2.11 Summary

This chapter starts with the detailed discussion on data warehouse and the necessity for its scalability. Here different warehouse architectures are represented in detail – some of which are centralized and some are distributed. These architectures give the inspiration to form the basis of our work for managing the distributed data warehouse. A comparison among shared memory, shared disk, and shared nothing multiprocessor systems is discussed, of which shared nothing is chosen in our work because of its scalability. Data warehouse modeling techniques, where dimensioning and schema designing are focused here. The main focus of our work is the parallel construction of virtual data cube from a distributed compressed data warehouse. Therefore, multidimensional data cube and its utility in Online Analytical Processing are discussed in this literature. Finally different compression techniques and parallel environment of distributed data warehouse are depicted here in detail.

Chapter 3
System Architecture

This chapter describes the detail analytical model with complete elaboration for the distributed data warehouse management in a parallel environment using compressed relational structure. According to the system overview mentioned in section 1.3, the overall system is partitioned into four tiers. Therefore this chapter starts with the data warehouse server where the source relation is laid in. Then it's method of compression and distribution to multiple servers is described in detail. The data cube is constructed in parallel directly from this compressed and distributed relational structure, which belongs to the main focus of our work and clearly depicted here. Finally this chapter is concluded with the operational management of the user query imposed onto the data cube.

3.1 Data Warehouse Server

After collecting data from external sources, it is extracted, transformed, loaded, refreshed and stored into the data warehouse server (Fig. 3.1). The metadata is incorporated into the data warehouse through the monitoring and integration process. Data marts are generated by partitioning the warehouse data based on the date, business type, geography, etc. They are the subset of corporate-wide data that is of value to a specific group of users. Its scope is confined to specific, selected groups, such as marketing data mart. A data mart can itself be a data warehouse.

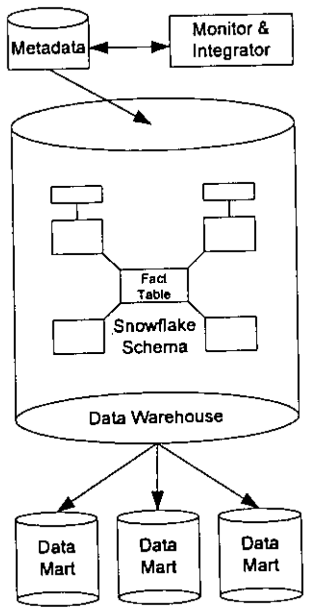


Fig. 3.1: Data Warehouse Server

When a data warehouse has to be built for an organization, we have to define the business requirements and agreed upon the scope of our application, and created a conceptual design. Then we translate our requirements into a system deliverable. To do so, we create the logical and physical design for the data warehouse.

The logical design is more conceptual and abstract than the physical design. In the logical design, we look at the logical relationships among the objects. In the physical design, we look at the most effective way of storing and retrieving the objects as well as handling them from a transportation and backup/recovery perspective. In this section, we concentrate on the logical design for the distributed data warehouse management in a parallel environment.

The process of logical design involves arranging data into a series of logical relationships called entities and attributes. An **entity** represents a chunk of information. In relational databases, an entity often maps to a table. An **attribute** is a component of an entity that helps define the uniqueness of the entity. In relational databases, an attribute maps to a column. Therefore our logical design results in (i) a set of entities and attributes corresponding to fact tables and dimension tables and (ii) a model of operational data from our source into subject-oriented information in our target data warehouse schema.

Data in the data warehouse must be structured using a schema. A schema is a collection of database objects, including tables, views, indexes, and synonyms. We can arrange schema objects in the schema models designed for data warehousing in a variety of ways. Most data warehouses use a dimensional model.

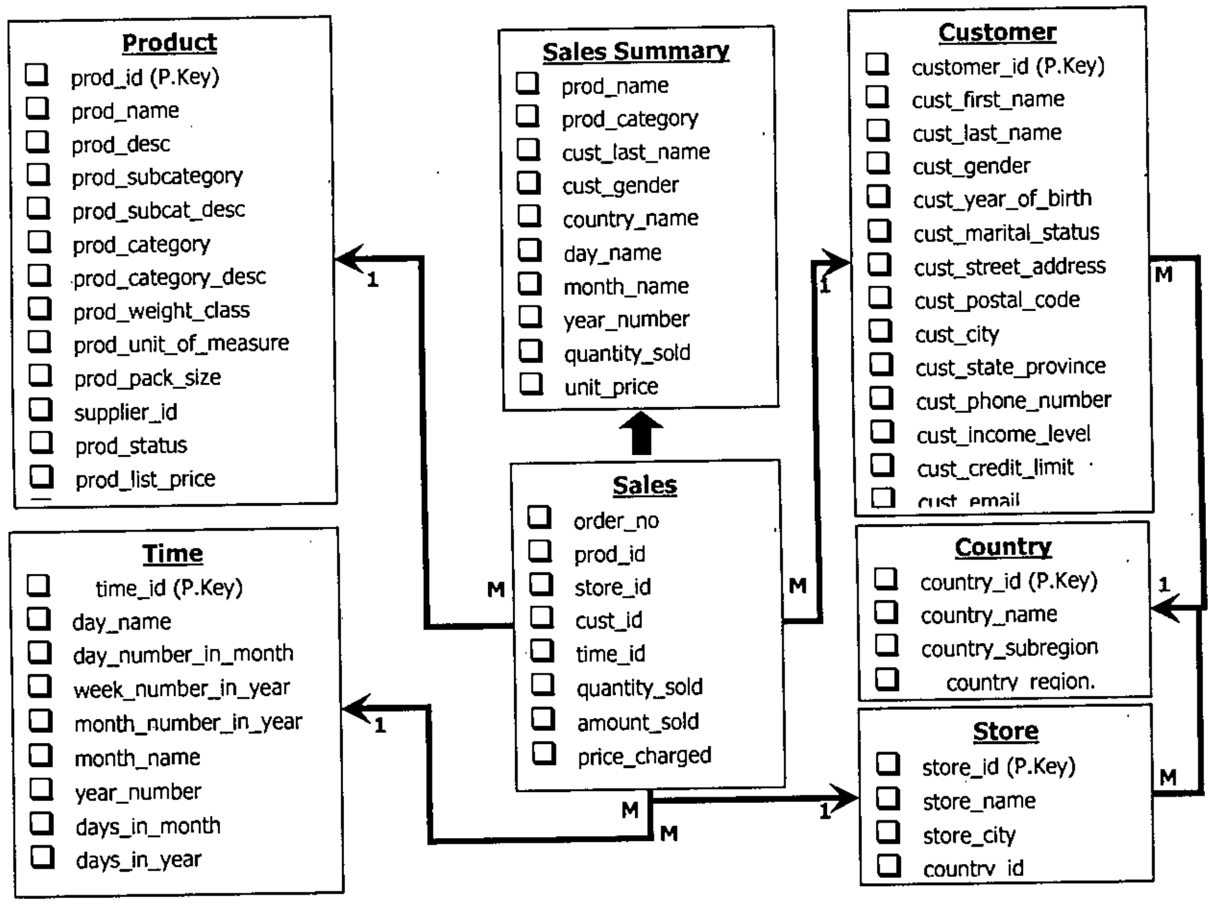


Fig. 3.2: Snowflake schema of a retail data warehouse

The physical implementation of the logical data warehouse model may require some changes to adapt it to the system parameters — size of machine, number of users, storage capacity, type of network, and software. However, among all the data warehouse schemas we have chosen the snowflake schema. The snowflake schema is a more complex data warehouse model than a star schema. However it inherits some properties of star schema, therefore we also get the advantages of star schema mentioned in section 2.6.2. It is called a snowflake schema because the diagram of the schema resembles a snowflake. Snowflake schemas normalize dimensions to eliminate redundancy. That is, the dimension data has been grouped into multiple tables instead of one large table. For example, a product dimension table in a star schema might be normalized into a products table, a product_category table, and a product_manufacturer table in a snowflake schema. While this saves space, it increases the number of dimension tables and requires more foreign key joins. The snowflake schema of classical retail data warehouse is portrayed in Fig [3.2]. This is collected from the oracle data warehouse corporation [30] and is used through out our work including the experiments.

A fact table typically has two types of columns: those that contain numeric facts (often called measurements), and those that are foreign keys to dimension tables. A fact table contains either detail-level facts or facts that have been aggregated. Fact tables that contain aggregated facts are often called summary tables. A fact table usually contains facts with the same level of aggregation. Though most facts are additive, they can also be semi-additive or non-additive. Additive facts can be aggregated by simple arithmetical addition. A common example of this is sales. Non-additive facts cannot be added at all. An example of this is averages. Semi-additive facts can be aggregated along some of the dimensions and not along others. An example of this is inventory levels, where you cannot tell what a level means simply by looking at it. From a modeling standpoint, the primary key of the fact table is usually a composite key that is made up of all of its foreign keys.

Dimension tables, also known as lookup or reference tables; contain the relatively static data in the warehouse. Dimension tables store the information we normally use to contain queries. Dimension tables are usually textual and descriptive and we can use them as the row headers of the result set. A dimension is a structure, often composed of one or more hierarchies, that categorizes data. Dimensional attributes help to describe the dimensional value. They are normally descriptive, textual values. Several distinct dimensions, combined with facts, enable you to answer business questions. Commonly used dimensions are customers, products, and time. Dimension data is typically collected at the lowest level of detail and then aggregated into higher level totals that are more useful for analysis. These natural rollups or aggregations within a dimension table are called hierarchies.

Hierarchies are logical structures that use ordered levels as a means of organizing data. A hierarchy can be used to define data aggregation. For example, in a time dimension, a hierarchy might aggregate data from the month level to the quarter level to the year level. A hierarchy can also be used to define a navigational drill path and to establish a family structure. Within a hierarchy, each level is logically connected to the levels above and below it. Data values at lower levels aggregate into the data values at higher levels. A dimension can be composed of more than one hierarchy. For example, in the product dimension, there might be two hierarchies--one for product categories and one for product suppliers. Dimension hierarchies also group levels from general to granular. A level represents a position in a hierarchy. For example, a time dimension might have a hierarchy that represents data at the month, quarter, and year levels. Levels range from general to specific, with the root level as the highest or most general level. The levels in a dimension are organized into one or more hierarchies.

Now as a matter of fact, a fact table can be grown very large which would not be suitable for the query processing. Hence the concept of materialized view or summary management comes, which is fairly straightforward. The idea is that we store summary data, or pre-computed results, instead of detailed data, in our data warehouse, for typical queries and reports. The result of the query for example, - "What were sales in the west and south regions for the last three quarters?" can be swiftly processed from the summary table, 'sales per month by region' than that of the 'sales' fact table, due to its much smaller size (Fig. 3.3). Materialized view has the advantages of faster and robust warehouse query processing. The optimizer transparently redirects the queries to the summary table. It also supports hierarchical rollup (ex. day -> month -> year). It optimizes frequent joins on complex schemas with selective filtering and subset of joined tables.

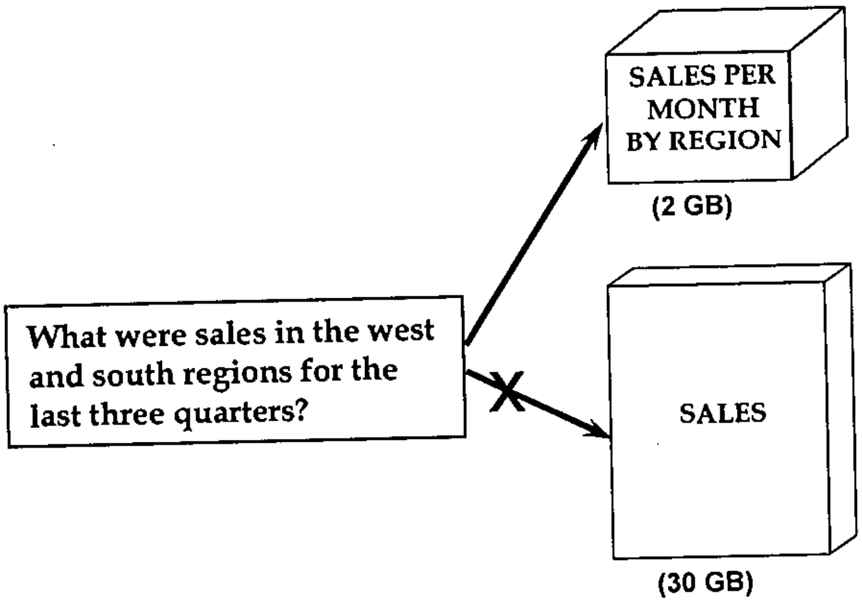

SALES PER MONTH BY REGION
(2 GB)
What were sales in the west and south regions for the last three quarters?
X
SALES
(30 GB)

Fig. 3.3: Example query onto fact and sales summary table

An example query is shown in Fig. 3.4, which is imposed onto the fact table. Therefore it is the duty of the query optimizer to transparently rewrites query on detailed data in some fashion to access data in the materialized view.

```
SELECT p.brand, r.country, t.month, SUM(s.amt) tot_sales
FROM sales s, region r, time t, product p
WHERE s.region_id = r.region_id
        AND   s.sdate = t.curdate
        AND   s.prod_code = p.prod_code
GROUP BY p.brand, r.country, t.month
HAVING SUM(s.amt) > 5000000;
```

```
SELECT brand, country, month, tot_sales
FROM        sales_sumry
WHERE tot_sales  > 5000000;
```

Fig. 3.4: Query rewrites by the query optimizer

3.2 Distributed Multi-Block Vector Architecture

With the intension of compressing the warehouse data, we apply our Distributed Multi-Block Vector Architecture (DMBVA) to store the relational schema in a compressed and distributed format. DMVBS is an extended modification of CMBVS [4]. Here for successful storage management, the relational warehouse is compressed and distributed over multiple column servers as shown in Fig. 3.5. Here each column server is associated with a domain dictionary, a large compressed vector and an index structure. Later on data cube is generated into the column servers in a distributed parallel environment. Application server stores the measure attributes and synchronizes among those column servers.

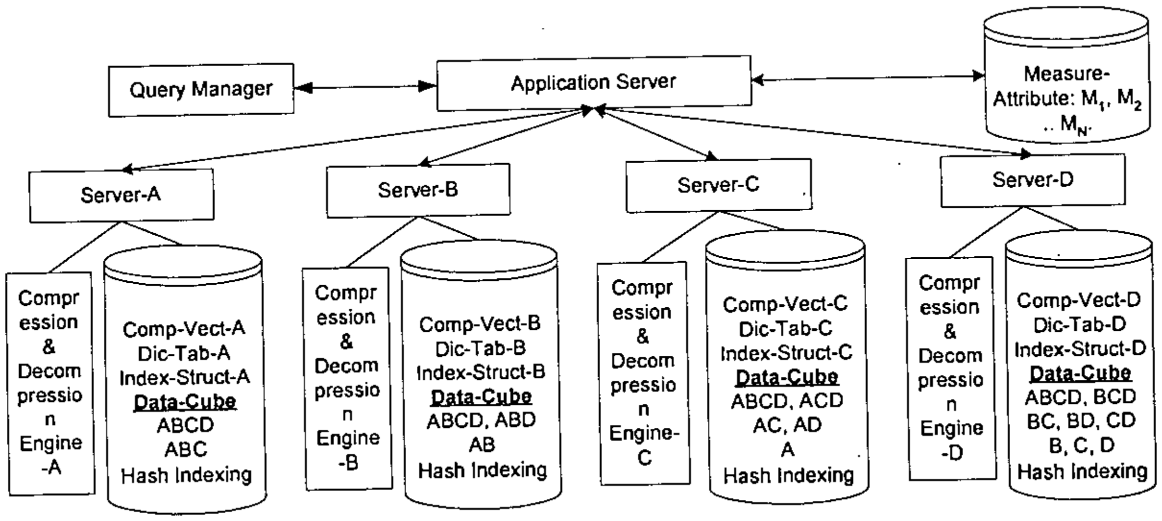


Fig. 3.5: General architecture of distributed compressed data warehouse containing a fact table of four dimensional attributes 'ABCD' and one measure attribute 'M'.

3.2.1 Domain Dictionary

The domain dictionary consists of set of lexemes with corresponding token values. It is updated dynamically whenever facing a new lexeme. We have created disk-based multiple part dictionaries to accommodate huge quantities of lexemes, where some parts resides in the memory and others are in the disk, which are fetched on demand basis. There are two main operations in the dictionary. *(i) getToken(lexeme):* This operation return the token of the given lexeme. If the lexeme is not present in dictionary, then add it there and return the corresponding new token. *(ii) getLexme(token):* It returns the corresponding lexeme of the token. The size of the resultant dictionary in the i^{th} column server is:

$$C_i \times L_i \tag{3.1}$$

Where, C_i =cardinality of domain attribute, L_i=average length of lexeme, I =index structure storage size. If p is the number of column servers, then total dictionary size:

$$\sum_{i=1}^{p} (C_i \times L_i \ ) \tag{3.2}$$

Hashing technique is applied to search the dictionary in the i^{th} column server, Therefore the searching time:

$$O(1) \tag{3.3}$$

3.2.2 Compressed Vector

A compressed vector represents a single column of relational database. Here only the tokens of the corresponding lexemes are stored and hence compressed. As the main memory is limited, the entire vector could not be accommodated in memory. So each vector is partitioned into blocks and a certain number of blocks reside in memory, the others are kept in the disk. Any operation in the vector ultimately propagated to the block. Multi-block structure reduces reorganization cost of vector and also reduces the wastage of space. Fig. 3.6 shows the structural design of compressed vector.

Here a vector consists of k×m blocks, among which m blocks can reside in memory. Now, a block is the container of some fixed number of compressed elements (tokens). Actually a block is a collection of some machine words that accommodate the compressed tokens. The bit string of the elements is mapped to the underlying word units contiguously.

If the bit string of the last element of a word can not be accommodated then, fraction of the bit string is stored in that particular word and the rest are shifted to the next word. Therefore, there is no wastage of space in the block except at the block boundary.

There are some significant features of multi-block structure: Blocks can be added or removed to accommodate addition, or restructuring of the vector. During evolution of the database, elements of some attributes may need 'broadening'. Each block can be adjusted dynamically and incrementally to accommodate changes in the size of element being stored in it. This block-at-a-time reorganization is the key to enabling the DBMS designer to meet deterministic maximum update time constraints, and hence to achieve scalable update performance.

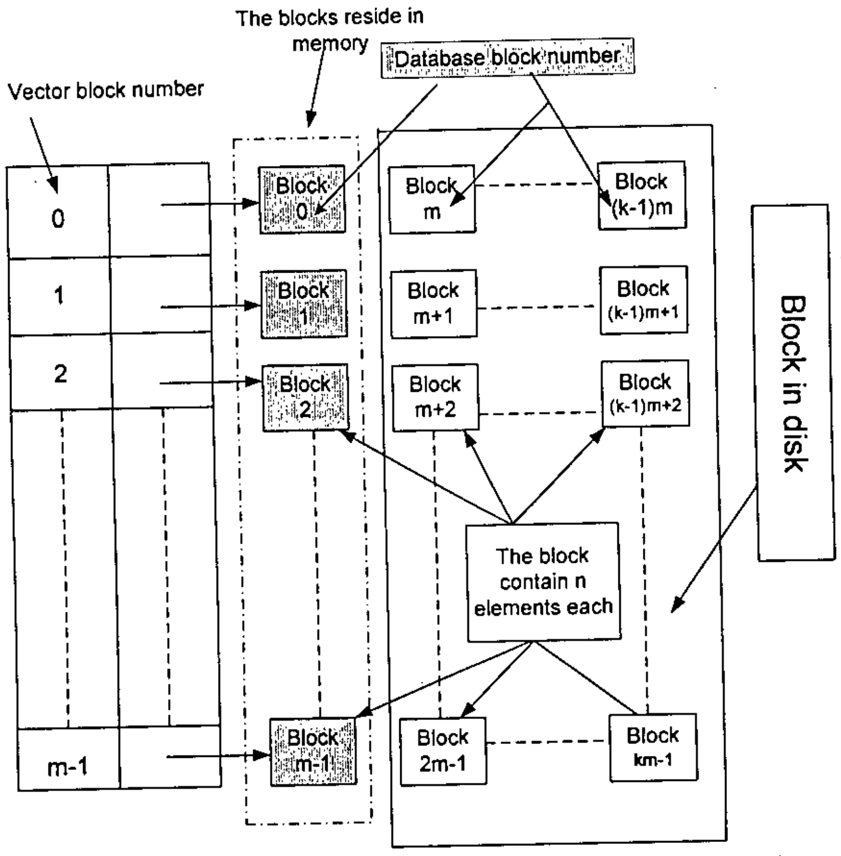


Fig. 3.6: Structural design of DMBVA compressed vector

Without this independence of update for individual blocks, the storing of a longer identifier would require adjustments of the vector width (and hence copying) of the entire storage of the vector, a task proportionate to the size of the vector. This is one of the most important aspects in the use of this structure in database systems. The independence of each block permits data compression to be applied within the block.

The main operations of the block are, ***(i) isEmpty():*** check whether the block is empty. ***(ii) addToken(token):*** insert a new token to the block. The insertion always occurs at the end of the block. The element size of the inserting element should be same as the element size of the block. If the element size of the token is larger than the element size of the block then widening operation is issued. ***(iii) getToken(index):*** This operation returns the ith token from the block and i must be smaller than or equal total number of elements in the block. ***(iv) widen(numBit):*** Widening operation is issued when the element-size of inserting token is larger than that of existing element-size. During the widening operation, all the elements in the block are reorganized and element-size is set to new element-size. To perform this operation, all the existing tokens are inserted to a temporary wordlist with new element size then the new element is added to the temporary wordlist. The old wordlist is deleted and the properties of the block are set with new element-size. ***(v) getTupleId(token):*** This operation returns a set of record-ids from the block that has the same token found in parameter. This record-id represents the index of matching tuple within the block (not the database record-id). If no matching token found in the block then it returns –1.

3.2.2.1 Fixed Number of elements per block (Variable storage size)

Here each block contains a fixed number of elements and hence the memory requirement for different blocks is different and depends on the size of elements in that block. This may be fixed when the system designer fixes the number of elements per block. In practice it does not appear to be a parameter to which the system's performance is sensitive. To achieve the full space utilization the number of elements in each block is to be a multiple of word size. This decision also reduces the computational complexity. The size of element stored in the vector is uniform within any block, but may vary between blocks. This arrangement though slightly less convenient to manage in terms of memory allocation provides direct addressability to the i^{th} element by means of a simple some operations.

3.2.2.2 Element to Word Mapping

There are two possibilities of element to word mapping:

- Map the bit string contiguously to the underlying word units (overlapping elements).

- Map the elements of the vector to words in such a way that no vector element ever overlaps a word boundary of underlying hardware (non-overlapping elements).

The second option appears to first sight to have a lower level of complexity since element will be sought only within one hardware defined word. It does, however, suffer from wastage of space at the word boundary. Depending on the number of elements in the block, there is another space overhead at the boundary of the block. In overlapping Element Mapping no space will be wasted at the word boundaries. But there is still wastage of space at the block boundaries. This wastage ranges from 0 to a maximum of (*wordSize*-1). We use the overlapping technique for our thesis considering space efficiency. Therefore, the overall size of the compressed vector is:

$$w\sum_{i=1}^{\lceil n/m \rceil}\left\lceil \frac{\mathrm{m}\times v_i}{w} \right\rceil \tag{3.4}$$

Here, n = total number of elements in the vector, m = total number of elements in a block, v_i = element size in the ith block (bits), w = word size (bits).

The overall creation time complexity is the sum of the,

(i) Overall widening operation (U=$O(\lceil n/m \rceil \times \log_2 m)$).

(ii) Time to store the blocks in disk:

$$O(\lceil n/m \rceil \times T_s) \tag{3.5}$$

(iii) Dictionary creation time: *total dictionary search time* + *total block storage time* =

$$O(C_i) + O(B_d \times T_d) \tag{3.6}$$

Here, T_S = time to store a compressed block into the disk, T_d = time to store a dictionary block into the disk and B_d = size of the dictionary block. Here, the network delay will be zero, because each compressed vector and its corresponding domain dictionary resides in the same column server. The flow chart for inserting tokens into vectors is shown in Fig 3.7.

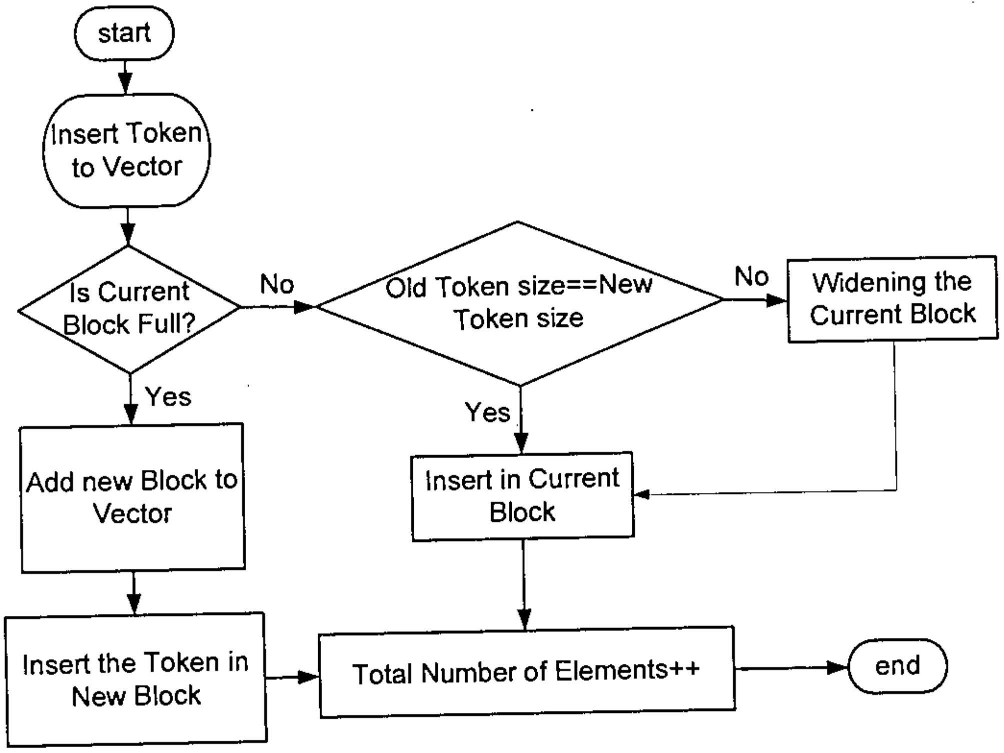

start
Insert Token to Vector
Is Current Block Full?
No
Old Token size==New Token size
No
Widening the Current Block
Yes
Yes
Add new Block to Vector
Insert in Current Block
Insert the Token in New Block
Total Number of Elements++
end

Fig. 3.7: Flow chart for inserting a token into a compressed vector.

3.2.2.3 Widening Operation

It requires an additional temporary storage because the existing elements need to add to temporary block with new element size. After the completion of widening the storage will be released.

$$\text{Temporary storage} = O\left(w\left(\left\lceil \frac{\mathrm{m} \times v_i}{\mathrm{w}} \right\rceil\right)\right) \tag{3.7}$$

Here, m = total number of elements in a block, v_i = width of each element in the ith block (bits) and w = word size (bits).

3.2.3 Index Structure

An index file is created if the lexeme is a search key. The index structure is associated to <lexeme, token> instance. Then we transform lexeme to token using dictionary and then token is used as a new search key. A hashing technique is used to extract the specific token from the multi-block vector structure.

3.3 Parallel Construction of Virtual Data Cube

A data warehouse is based on a multidimensional data model which views data in the form of a data cube. A data cube, such as sales, allows data to be modeled and viewed in multiple dimensions. It resembles dimension tables, such as item (item_name, brand, type), or time (day, week, month, quarter, year) and fact table which contains measures (such as dollars_sold) and keys to each of the related dimension tables. An example data cube is shown in Fig. 3.8.

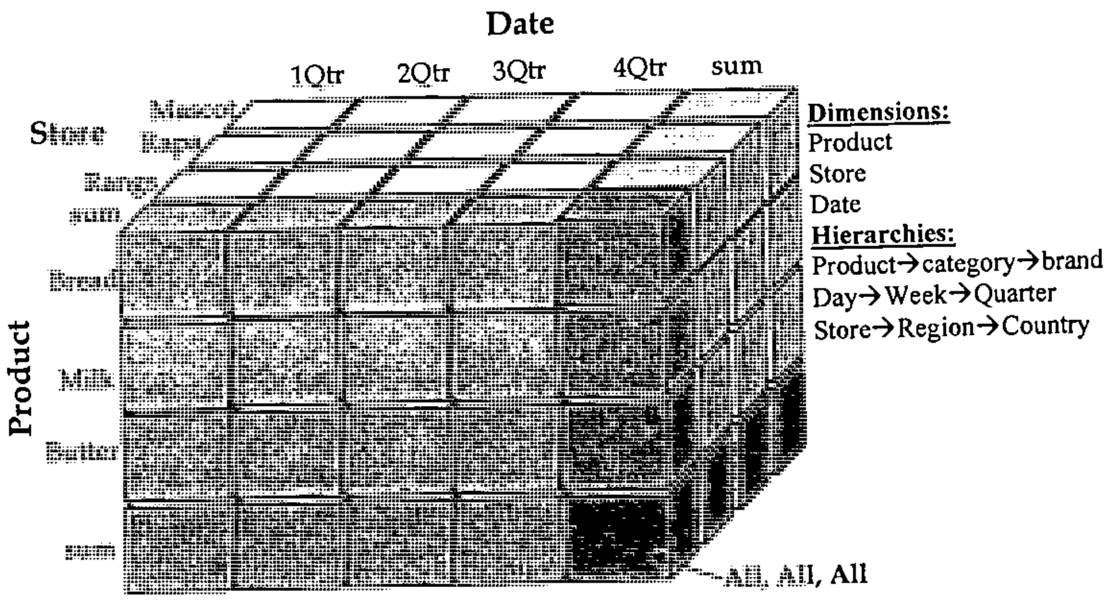


Fig. 3.8: Multidimensional data cube

Data cube is the computation of aggregation on all possible combination of dimension attributes. For a base relation R with N dimension attributes $\{D_1, D_2 \ldots, D_N\}$ and one measure attribute M, the full data cube of R on all the dimensions is generated by CUBE BY on $D_1, D_2 \ldots, D_N$. it is equivalent to the union of 2^N distinct GROUP BY. The CUBE BY is a multidimensional extension of the relational operator GROUP BY, which computes GROUP BY on all possible combinations of grouping attributes in the CUBE BY clause. Each group-by is called a cuboid. The CUBE BY is a very expensive operator, and the size of its result is extremely large especially when the number of CUBE BY attributes and the number of tuples in the source relation is large. Data cube can be viewed as a lattice of cuboids. The bottom-most cuboid is the base cuboid, the top-most cuboid (apex) contains only one cell and it is already figured out in section 2.7. The number of cuboids in an n-dimensional cube with L levels is:

$$T = \prod_{i=1}^{n} (L_i + 1) \tag{3.8}$$

Among the relational OLAP (ROLAP) and multidimensional OLAP (MOLAP) we have chosen the ROLAP, because here the cuboids are conventional database tables, we can process and query them using traditional RDBMS techniques, such as indexes and joins. This format is likely to be efficient for large data collections, since the tables must include only data cube cells that actually contain data.

3.3.1 Data Cube File Format

Data cube construction format is depicted in Fig. 3.9. Here the smallest unit is the record_ID, which is a sequence of bit string. Every single record_ID and its corresponding measure_Value melded into a cube_Record. Multiple cube_Record form a cube_Word, where multiple cube_Word fused into a cube_Block. Every cube_Block is independently addressable so the structure is suitable for large data cube. Here each cube_Block contains a fixed number of cube_Records and hence the memory requirement for different blocks of different cuboids may be varied. However the cube_Records of a particular cuboid are mapped contiguously to the underlying cube_Words (overlapping phenomena). So no space is wasted at the word boundaries. But there is still wastage of space at the block boundaries. This wastage ranges from 0 to a maximum of (size of cube_Word-1). Then, multiple cube_Block form a cube_Vector, alternatively which is called a cuboid. As the main memory is limited, the entire cuboid may not be accommodated into memory. So a certain number of cube_blocks are resided in the memory, the others are kept in the disk. Finally all the cuboids collectively form the resultant data cube.

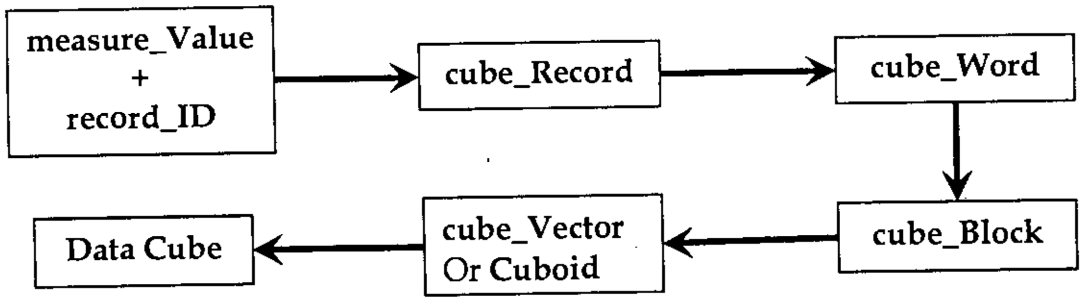


Fig. 3.9: Data Cube construction procedure.

To generate a child cuboid, for example ABC, we store the contents like: $A_1B_1C_1F_1$, $A_1B_1C_2F_2$, ..., $A_{|A|}B_{|B|}C_{|C|}F_n$. Here, $A_xB_yC_z$ is the record_ID, F_{val} is the corresponding measure value or fact. A_x implies the bit string of the token x from the dimension A. This is clarified in Fig. 3.10.

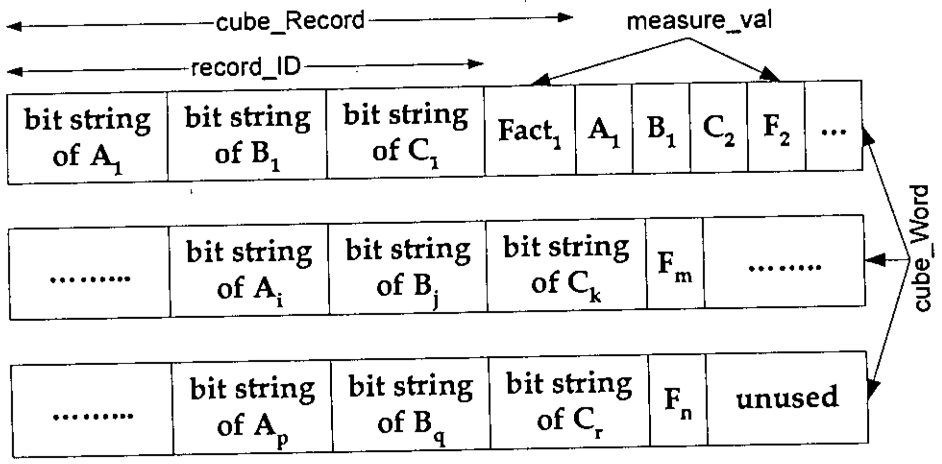


Fig. 3.10: Relation among the data cube components

We have generated the data cube directly from the compressed vector in parallel. The cardinality of each dimension attribute domain is known in advance, and each attribute value is one-to-one mapped to a token value. The token value 'zero' is reserved for *ALL* value in the GROUP BY operation. A data cube is constructed from a schedule tree, whereas a schedule tree is generated from a cuboid lattice by solving a minimum spanning tree problem. This is illustrated in Fig. 3.11 for a four dimensional attribute 'ABCD'. Using this schedule tree, all the cuboids are generated eventually. Here we assume that $|A| \geq |B| \geq |C| \geq |D|$. Fig. 3.11 (a) portrait the cuboid lattice. There are two types of data cube file: root cuboid and child cuboid. Fig. 3.11 (b) depicts the flow of data cube generation.

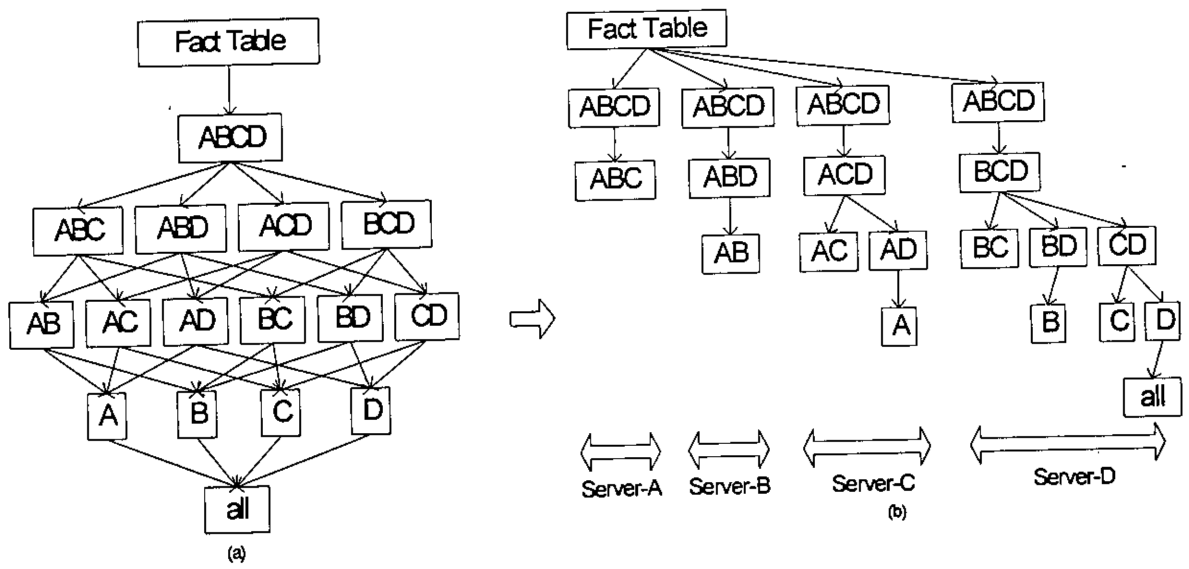


Fig. 3.11: (a) A cuboid lattice of four dimensional attributes 'ABCD'. (b) A Schedule tree produced from the aforementioned lattice

Here the root cuboid 'ABCD' is generated from the compressed fact table by the servers A, B, C and D in parallel. Later on, the child and interior cuboids are generated from their parent cuboids by the individual servers in parallel. A large data cube file is partitioned into blocks based on the memory capacity for better portability and I/O channel utilization. The contents of root cuboid are not ordered according to the construction algorithm, a hashing technique is used here at the times of query. But the contents of the child cuboids are ordered based on record-IDs. Binary search on a particular cluster is applied here at the time of query.

3.3.2 PCVDC Algorithm

Steps for Parallel Construction of Virtual Data Cube (PCVDC) are as follows:

Step 1: Starting from the first token, each column server extracts a chunk of tokens from their individual compressed vector in parallel and sends it to the application server.

Step 2: Application server then generate the record_ID's from the received tokens.

Step 3: Application server then broadcast {record_ID, measure_val} to all the column servers.

Step 4: Each column server inserts the received record_ID & measure_val into a block of the cube_vector..

Step 5: Repeat step 1 to 4 until the last token is extracted. (Hence, the root cuboid is generated

Step 6: Each column server locally computes the child cuboids from its smallest parent cuboid in parallel as assigned in the schedule tree.

Step 7: Repeat step 6 until the complete data cube is constructed.

The detailed algorithms are depicted below, where Algorithm 3.1 describes the data cube construction procedure participated by the Application Server, Algorithm 3.2 describes the data cube construction procedure participated by each of the Column Server in parallel, and finally from root-cuboid to child-cuboid construction sub-procedure by each of the Column Server in parallel is written in Algorithm 3.3.

Algorithm 3.1: PCVDC in the Application Server

```
Input:     tokens from column server
Output:    record_IDs and measure_Values
1.   create a server socket
2.   create as many as child threads for each column server
3.   pass socket acceptance to each thread constructor
4.   for each column server thread pardo
5.     while(true) do
6.         wait to receive a chunk of tokens from each column server
7.         send this token to the root thread
8.         root thread generates the record_IDs using the collected tokens and extract the
           corresponding measure values
9.         send these record_IDs and measure_Values to the column server
10.        if the last token had received then break
11.    end while
12.  end for
```

Algorithm 3.2: PCVDC in the Column Server

```
Input:     schedule tree, compressed vector file
Output:    complete data cube
1. create a thread for this column server
2. create a socket and connect to the server port
3. while EOF of all compressed vector file is not reached do
4.   extract a chunk of tokens from the vector
5.   send these tokens to the application server
6.   wait to receive the record_IDs and the measure_Values from the application server
7.   store them to the rootCuboid file
8. end while
9. constructChildCuboid(rootCuboid)
```

Algorithm 3.3: Constructing Child Cuboid

```
Input:     schedule tree, rootCuboid
Output:    child cuboids
1. r ← rootCuboid
2. generate all childCuboids of r
3. for each children c of r from left to right do
4.   if c has no children then
5.     return to parent cuboid
6.   else constructChildCuboid(c)
7.   end if
8. end for
```

3.3.3 PCVDC Analysis & Evaluation

Let, m = average cardinality of each dimension attributes. n = Average number of tuples (elements) in compressed vector. d = Number of dimension attributes. b = Total number of blocks.

- For a ***single server system***, time complexity to construct a data cube:

$$O\left(m^{d}\left\{ n + \frac{n}{m}(d-1) \right\} \right) + O\left({}^{d}C_{d-1}m^{d-1+1} + {}^{d}C_{d-2}m^{d-2+1} + \cdots + {}^{d}C_{1}m^{1+1} \right) = O\left(m^{d}n\right) + O\left(dm^{d}\right) = O\left(m^{d}n\right) \tag{3.9}$$

In the case of root cuboid generation: Total number of tuples in the data cube is m^d, For each tuple in the data cube, we have to sequentially search all the n tuples of the first dimensional attribute (vector) to match with the first element of that data cube tuple. If a match is found then we have to extract the same tuple of the next dimensional attribute (vector) in order to find a match with the next element of that particular tuple in the data cube. On an average, there are n/m replicas of each tuple of the domain dictionary in the vector. So for each tuple in the data cube, total searching for the rest of the d-1 dimension attributes (vectors) is:

$$\frac{n}{m}(d-1) \tag{3.10}$$

In the case of child cuboid generation: Since there are total ${}^{d}C_{d-x}$ '*d-x*' dimensional child cuboids of a d dimensional cuboid, for each of this child cuboid, a total of m^{d-x+1} comparison is made.

- Now, for a ***multiple server system***, time complexity to construct a data cube:

$$O(n) + O\left(m^{d}\right) = O\left(m^{d}\right) \tag{3.11}$$

Here to generate the root cuboid, only n number of searching is made in parallel through the compressed vectors. Whereas, for child cuboid generation, it is same as single server system, but the generation is distributed among d number of servers. So a maximum of m^d comparison is made. The maximum size of the data cube is:

$$\sum_{c=1}^{2^N}\left\{ \left(\prod_{j=1}^{d_c} \left(C_{cj}+1\right) \right) \times \left(B_c + M\right) \right\} \tag{3.12}$$

Here, number of dimensions in the source relation is N, so total number of cuboids is 2^N, number of dimension in the cth cuboid is d_c, cardinality of the jth domain of the cth cuboid is C_{cj}, size of the block-ID is B_c and measure value is M in cth cuboid.

3.4 Optimality Issue of PCVDC

Prefix and suffix redundancy are identified in our approach and eliminated as well. Prefix redundancy is understood by the fact that there is a cube with dimensions A, B and C. each value of dimension A appears in 4 group-bys (A, AB, AC, ABC) and possibly many times in each group-by. This prefix redundancy can be further classified into inter-cuboid and intra-cuboid prefix redundancy. Since there is no direct storage of dimension attribute values, the prefix redundancy is automatically wiped out.

Suffix redundancy occurs when two or more group-bys share a common suffix (like ABC and BC). For example, consider a value B_j of dimension B that appears in the fact table only with a single value A_i of dimension A. Then, the group-bys $<A_i, B_j, x>$ and $<*, B_j, x>$ always have the same value for any value x of dimension C. More specifically taking an extreme case as an example, let a fact table R have only a single tuple $r = <A_2, B_1, C_3, M>$. Then, the data cube of R will have $2^3 = 8$ tuples, $<A_2, B_1, C_3, V_1>$, $< A_2, B_1, *, V_2>$, $<A_2, *, C_3, V_3>$, $<A_2, *, *, V_4>$, $<*, B_1, C_3, V_5>$, $<*, B_1, *, V_6>$, $<*, *, C_3, V_7>$, $<*, *, *, V_8>$. Since there is only one tuple in relation R, we have $V_1 = V_2 = \dots = V_8 = aggr(r)$. Therefore, we only need to physically store one tuple $<A_2, B_1, C_3, V>$, where $V= aggr(r)$. So the cube for R with 8 tuples can be condensed into one tuple. Our optimization feature detects and purges this type of redundancy. Algorithm 3.4 depicts the optimal cuboid construction procedure where the suffix redundancy is eliminated.

Algorithm 3.4: Constructing Optimal Cuboid

```
Input:     name: cuboid name = 'X1X2...Xn tokens from column server
           metadata: name.dat = 'X1X2...Xn'.dat
Output:    optimal cuboid
1.  Load 'X1X2...Xn'.dat into the memory.
2.  t ← 1st tuple going to be added into the cuboid 'X1X2...Xn'
3.  while any tuple remains to be added do
4.        bID ← blockID of t
5.        mVal ← measureValue of t
6.        check the existence of t into the metadata
7.        if not found
8.              add the bID and mVal into the cuboid
9.        end if
10.       if $Xj_{valj} \ldots Xk_{valk}$ is a constant suffix of $Xi_{vali} \ldots Xj\text{-}1_{valj-1}$, where i <= j <= k, then
11.             broadcast (fileName=Xj...Xk, suffixToken=valj...valk, prefixName=
          Xi...Xj-1, prefixToken= vali...valj-1) targeted to the column servers
          which contain any cuboid, whose name starts with Xj...Xk.
12.       end if
13.       t ← next tuple going to be added into this cuboid
14.  end while
```

Algorithm 3.5: Update Metadata

(Running under an infinite thread on each column server)

```
Input:     broadcastInfo
Output:    updated metadata file
1.  if the file addressed in the broadcastInfo do not exist
2.        then create this file.
3.  else if the broadcastInfo is already exist
4.        then return
5.  else
6.        add 'suffixToken, prefixName, prefixToken' into the file referred as fileName.dat
```

The general idea is, while generating any cuboid, if there is any possibility of suffix redundancy then the suffix and prefix related information is broadcast to each of the column server to update their metadata file with the broadcast information if they are destined so. Later on, those redundant tuples would not be further generated by the destined column server. This is because, while constructing their individual cuboid, its associated metadata is loaded into the memory, and those redundant tuples would be excluded in the resultant cuboid.

For updating the metadata, a thread is constantly running in each column server to receive the broadcast information and update its metadata as well if they are destined so. This metadata updating algorithm is reported in Algorithm 3.5.

3.5 Query from the Data Cube

For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses. In a distributed system, other issues must be taken into account:

- The cost of a data transmission over the network.
- The potential gain in performance from having several sites process parts of the query in parallel.

The relative cost of the data transfer over the network and data transfer to and from disk varies widely depending on the type of network and on the speed of the disk. Thus, we cannot focus solely on disks cost or on the network cost. Rather, we must find a good tradeoff between the two. A detailed query operation is shown in Fig. 3.12.

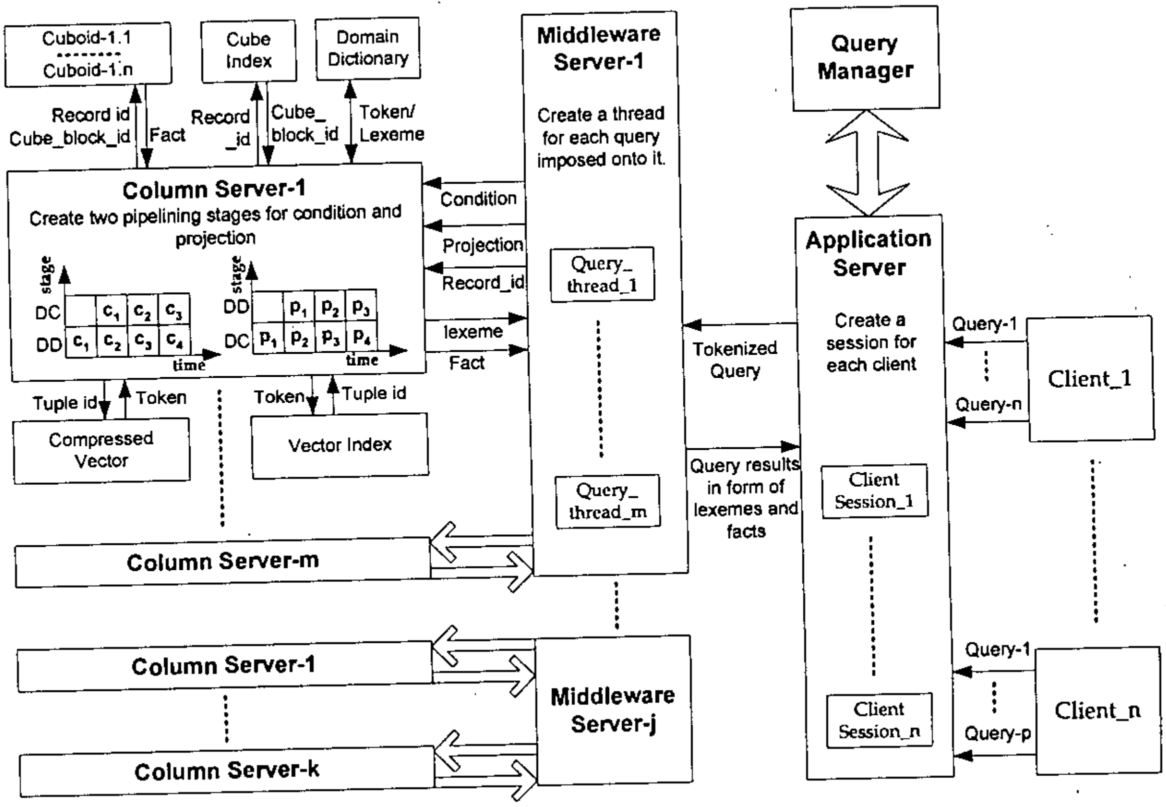


Fig. 3.12: Query architecture of PCVDC

Algorithm 3.6: Query Operation

Input: query
Output: resultant dataset

1. Application server creates a client session for each client when client connected to application server. The client session is created by multithreading.
2. Application server talk over the query manager for each query and the cheapest execution plan is found out among the many possible execution plans that give the same answer.
3. Translate each query into the SQL form.
4. Select the appropriate Middleware server from its metadata, which is associated with the column servers under which the query results reside.
5. Middleware server decomposes the tokenized query into individual projection, sort, where condition, join operation and determine the dependency between them. Then it forwards the request for projection and condition operation to the pertinent column servers.
6. Column servers extract the result of the condition and projection operation from the appropriate cuboids with the help of the cube index.
   6.1. For condition it creates two stages Pipeline for Domain Dictionary (DD) in the 1st stage and Data Cube (DC) in the 2nd stage. Here C_x implies condition number x.
   6.2. For projection it also creates two stages pipeline for Data Cube (DC) in the 1st stage and Domain Dictionary (DD) in the 2nd stage.
   6.3. When different pipelining stages try to invoke the same domain dictionary or data cube block, it is handled thread synchronization.
7. Finally, the results containing facts and lexemes are sent back to the Middleware server, which are packed with the query ID and eventually passed back to the Application server.

3.5.1 Time Complexity to Query from a Data Cube:

Let, t = Query tokenization time, m = Average no. of cube records per cube_block. d = Number of dimension attributes, and net_d = overall data transmission delay through the network. Now, the overall query response time = Query tokenization time + Cube_block search time + Cube_record searching time + Network delay. Since a hash indexing is used to locate the cube_block and binary search is applied to locate the cube_record, therefore cube_block searching time is $O(1)$ and cube_record searching time is $O(\log_2 m)$. So, the overall query response time we achieved is:

$$O(t + \log_2 m + net_d) \qquad (3.13)$$

3.5.2 Intraquery Parallelism

Intraquery parallelism refers to the execution of a single query in the parallel on multiple processors and disks. Using intraquery parallelism is important for speeding up long running queries. Interquery parallelism does not help in this task, since each query is run sequentially.

To illustrate the parallel evaluation of a query, consider a query that requires a relation to be stored. Suppose that the relation has been partitioned across multiple disks by ranges partitioning on some attribute, and the sort is requested on the partitioning attribute. The sort operation can be implemented by sorting each partition in parallel. Then concatenating the sorted partitions we get the final sorted relation.

Thus, we can parallelize a query by parallelizing individual operation. There is another source of parallelism in evaluating a query: The operator tree for a query can contain multiple operations. We can parallelize the evaluation of the operator tree by evaluating in parallel some of the operations that do not depend on one another. We chose the second option which is called interoperation parallelism. To evaluate an operator tree in a parallel system, we must make the following decisions:

- How to parallelize each operation and how many processors to use for it.
- What operations to pipeline across different processors, what operations to execute independently in parallel, and what operations to execute sequentially, one after another.

When the operations in a query expression depend on one another we apply the pipelined parallelism. If there is no such dependency we apply the independent parallelism. Query optimization is also taken into account in some cases which is performed by the query optimizer. Here a query is taken and the cheapest execution plan is found out among the many possible execution plans that give the same answer. Thus we speed up the processing of a query by executing in parallel the different operations in a query expression.

3.5.3 Interquery Parallelism

In interquery parallelism, different queries or transactions execute in parallel with one another. Transaction throughput can be increased by this form of parallelism. However, the response times of individual transactions are no faster than they would be if the transactions were run in isolation. Thus, the primary use of interquery parallelism is to scale up a transaction-processing system to support a larger number of transactions per second.

Supporting interquery parallelism is more complicated in a shared-disk or shared-nothing architecture. Processors have to perform some tasks, such as locking and logging, in a coordinated fashion, and that requires that they pass messages to each other. A parallel database system must also ensure that two processors do not update the same data independently at the same time. Further, when a processor accesses or updates data, the database system must ensure that the processor has the latest version of the data in its buffer pool.

In comparison with our system there is no update and delete operation in the data warehouse query operation. Moreover the resultant data cube is independently distributed and the operations from the different query expressions which are imposed onto the same cube_Record are managed by pipelining. Therefore is no necessity of locking and logging in the query operations and cache-coherency problem is eliminated as well.

3.5.4 An Example Query Operation

An example query operation is shown in Fig. 3.13, which is constructed from the schedule tree (Fig. 3.11(b)) of the PCVDC architecture. The example query is made based on customer relation entropy. Given five dimensional attributes, which are ID (X), Name (A), Street (B), City (C), Status (D), and one measure attribute named Salary (M). Let the query is "Select Name, Max (Salary) From Customer Relation Where Street = 'bijoy' AND Status = 'Unmarried'." This query is imposed to the query manager which is forwarded to application server in a different manageable format. Then it is partitioned into sub-query by the application server and send on to the appropriate column server for tokenizing the query attributes. After that, as the result of the query lies in the cuboid 'ABD', whereas the cuboid 'ABD' resides in the column server-B, so the query tokens are passed to column server-B. Then the result, Name = '011..0' and Salary = '40000' is passed back to application server. Token '011..0' is further sent to column server–C for decompression, therefore Name = 'Obayed' is returned, which is passed back to the query manager.

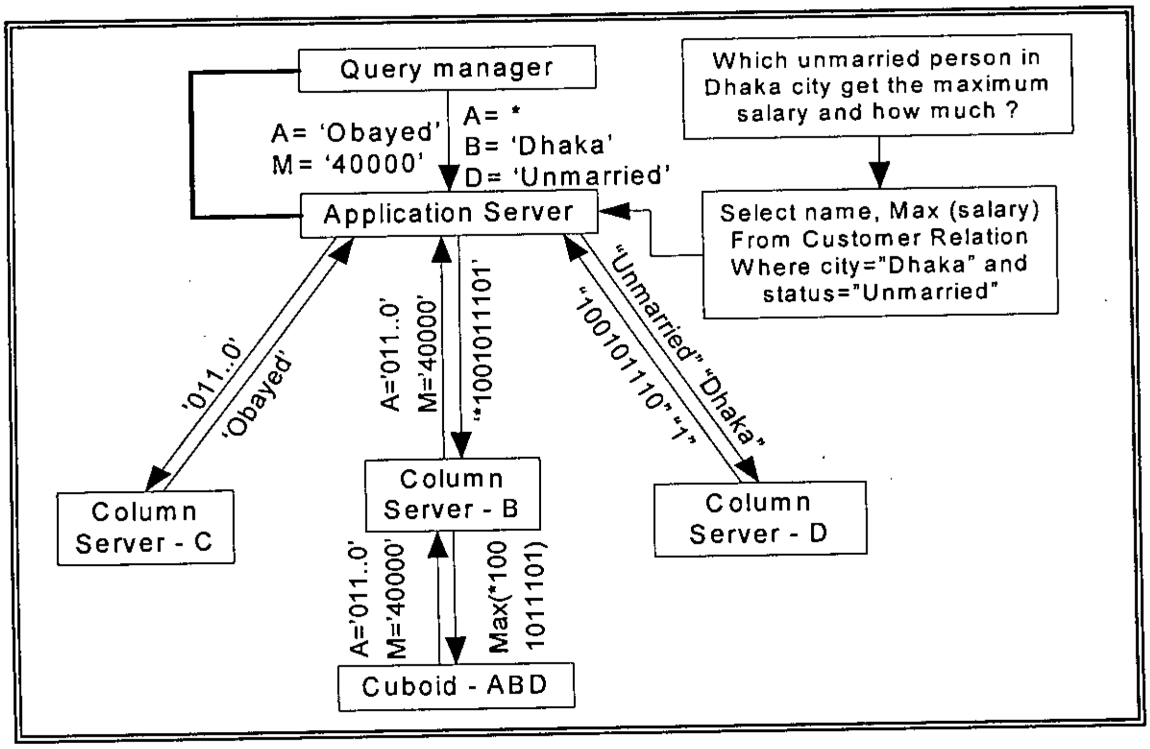


Fig. 3.13: A Query example

3.6 Summary

In order to achieve the maximum performance requirement and response times, parallel database system and also parallel algorithms are being vigorously exploited. Parallelism is appropriate for data warehouse because parallel systems can be constructed at a low cost without the need for any specialized technology, by using existing sequential computers and relatively cheap interconnection networks in today's environment. Complex queries, such as those involving joins of several tables or searches of very large tables, are best executed in parallel. With this intension, we developed a four tier architecture. There, the source of the data cube was compressed first. Afterwards, the compressed form of data was distributed and the data cube was generated directly from here without any decompression in a parallel environment. Furthermore optimality issues like prefix and suffix redundancy in the data cube are taken into consideration and eliminated as well. Hence, it took significantly reduced creation and storage cost for the data cube.

Chapter 4
Results and Discussions

Our target was to achieve maximum compression of the data warehouse, then its distribution for better manageability and parallel issue and finally a compact data cube construction from this compressed distributed architecture in parallel. Therefore, the objective of the experimental work is to verify the feasibility and scalability of the design of the distributed compressed multi-block vector, the dictionary and data model of the architecture. It also verifies the scalability, construction efficiency, query optimization from the virtual data cube. The experimental evaluation has been performed with large synthetic data. The storage of distributed compressed multi-block is compared with widely used Microsoft SQL Server. Also the storage, construction and query response time of the virtual data cube is compared with BSF, Dwarf and Cubetrees [15]. Throughout this chapter we call the components of our system - 'DMBVS' for Distributed Multi-Block Vector Structure and 'PCVDC' for Parallel Construction of Virtual Data Cube.

4.1 Experimental Setup

In our experimental setup there were one application server and several column servers (one for every compressed vector) connected via LAN. Each server having 3GHz Pentium IV processor, 40GB hard disk and 512MB RAM. The platform was windows XP and was implemented by java development kit 1.5.

4.1.1 Network Setup

We used TCP (Transmission Control Protocol) through the classes in java.net for the communication between two of our applications residing in different computers. TCP is a connection-based protocol that provides a reliable flow of data between two computers. Data transmitted over the network is accompanied by addressing information that identifies the computer and the port for which it is destined. The computer is identified by its 32-bit IP address, which IP (Internet Protocol) uses to deliver data to the right computer on the network. Ports are identified by a 16-bit number, which TCP use to deliver the data to the right application.

A server application binds a socket to a specific port number. A socket is one endpoint of a two-way communication link between two programs running on the network. This has the effect of registering the server with the system to receive all data destined for that port. On the client-side, the client knows the hostname of the machine on which the server is running and the port number to which the server is connected. To make a connection request, the client tries to rendezvous with the server on the server's machine and port as shown in Fig. 4.1.

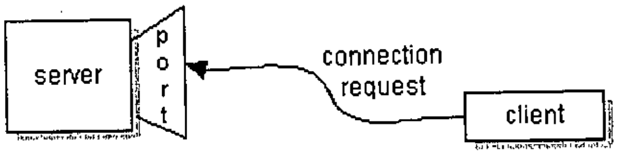


Fig. 4.1: Connection request from client to a server port.

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same port. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client (Fig. 4.2).

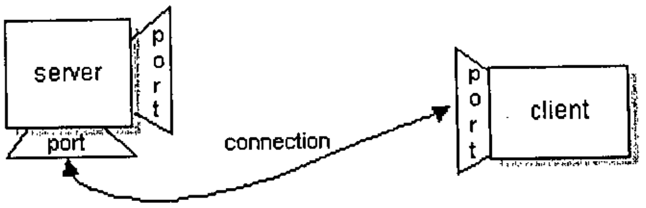


Fig. 4.2: Established connection between client and server.

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can then communicate by writing to or reading from their sockets. By using the java.net.Socket class, our java programs can communicate over the network in a platform-independent fashion. Additionally, java.net includes the ServerSocket class, which implements a socket that our Application and Middleware server can use to listen for and accept connections to clients.

During the root cuboid of the data cube construction process we used multicasting which can be thought of as broadcasting data over a network connection to many connected agents, as opposed to unicasting packets between two agents on a normal connection. Java.net includes a class called MulticastSocket. This kind of socket is used on the Column Servers to listen for packets that the Application Server broadcasts to multiple Column Servers.

4.1.2 Multithreaded Environment

To support concurrent access onto the Application server by multiple Column servers for the root cuboid generation we used multithreading. Also multiple tokenized query requests which are imposed onto the Column servers are handled by multithreading. Basic support for threads in the Java platform is in the class java.lang.Thread. It provides a thread API and all the generic behavior for threads. The behaviors include starting, sleeping, running, yielding, and having a priority. The execution of multiple threads on a single CPU, in some order, is called scheduling. Therefore thread scheduling is applied in our system to support the varieties of projection and multiple predicates select queries. The Java platform supports a simple, deterministic scheduling algorithm called fixed-priority scheduling. At any given time, when multiple threads are ready to be executed, the highest-priority thread is chosen for execution. A lower-priority thread starts executing only when that highest-priority thread stops or is suspended. When all the Runnable threads in the system have the same priority, the scheduler arbitrarily chooses one of them to run.

When the multiple query threads in our system tried to access the same data cube or dictionary block asynchronously, it was handled by Thread Synchronization. One thread got it, and the others went back to waiting. Furthermore the query threads were also able to notify one another when they finished their job. It was done by notifyAll method which woke up all threads waiting on the object. The awakened threads again compete for the lock.

Collection of threads is managed by java Thread Pool to improve the performance when executing large numbers of tasks as a result of reduced per-task invocation. We instantiated an implementation of the Java ExecutorService interface to use thread pool and then this thread pool was handed a set of tasks.

4.2 The Data Sets

We chose a classical retail data warehouse of an enterprise which is collected from the oracle data warehouse corporation [30]. A synthetic data generator was implemented to generate the data that resembles the real data sets for merchandising relation. We assumed five attributes in the relation that is given in the Table 4.1. We considered approximately 4 million customers of an enterprise. Based on that, the approximate cardinality of each dimension attribute is shown subsequently. Each record is estimated as 104 bytes in the uncompressed base relation, whereas each compressed record occupies at most 7 bytes.

Table 4.1: The sales relation structure

Attribute Name	Data Type	Cardinality	Length (bytes)	Bits required
Sales Id	Integer	Primary key	4	22
Product Name	Varchar	507	30	9
Customer Name	Varchar	5000	30	13
Customer Status	Varchar	2	10	1
Sales City	Varchar	64	30	6
Price Charged	Double	Measure value		
Total			104 bytes	51bits ≈ 7 bytes

4.2.1 Synthetic Data Generator

It is quite a hard task to gather the real data sets for the sales relation. We, instead deign a data generator that generates the data that resembles the real data sets for sales relation. The approximate cardinality of the domain values of each attribute is given in the Table 4.1. To create synthetic data, we generate a random number within a value equal to the domain cardinality of an attribute and write the number in a text file with a fixed width. Each line of the text file contains a tuple of a sales relation. The input manager load the text file to extract raw data and read each attribute value from the file and send the value to compression manager.

4.3 DMBVA Storage Efficiency

The storage space occupied by DMBVA demonstrates a significant compression factor against SQL Sever, and is shown in Table 4.2. The average compression ratio is 28.87 with respect to SQL Server. This is because SQL Server needs to store the original source relation in an uncompressed format with some additional information.

Table 4.2: Storage requirement of DMBVA vs SQL Server

Number of records (million)	Storage space in SQL Server (MB)	Storage space in DMBVA (MB)	Compression Factor against SQL Server
1	251.92	9.525	26.45
2	501.86	18.312	27.40
3	755.76	26.32	28.71
4	1011.65	33.56	30.14
5	1289.36	41.20	31.30

4.4 Data Cube Storage

The compressed vector act as the source relation for the generation of the data cube, and the data cube was generated directly from this compressed format in a parallel environment. Since the source relation was compressed, and the generated data cube was virtual (due to the direct absence of measure attribute), so a great deal of reduction in the ultimate storage space was achieved.

We compared PCVDC system with binary storage footprint (BSF) and Dwarf [15].. In our system, the fact table i.e. the compressed vector contained 100000 tuples with a uniform distribution of the dimensional values over a cardinality of 1000. The BSF representation models the storage required to store the views of the cube in non-indexed binary relation. In the Table 4.3, for different dimensions the data cube storage occupied by different methods is demonstrated. BSF comparatively took much more space for all the dimensions, because all the cuboids are stored here in non-indexed binary summary table without throwing out the redundancy. Excluding Prefix but Including Suffix redundancy (EPIS) both Dwarf and PCVDC perform better. However, in this case, PCVDC dominates Dwarf in the data cube storage reduction due to the absence of lexemes.

Table 4.3: Data Cube storage cost comparison with other system (Excluding Prefix but Including Suffix redundancy - EPIS)

Dimension	BSF	Dwarf EPIS	PCVDC EPIS
10	2333 MB	1322 MB	538.33 MB
15	106 GB	42.65 GB	21.57 GB
20	4400 GB	1400 GB	838.37 GB
25	173 TB	44.8 TB	30.83 TB
30	6.55 PB	1.43 PB	1.16 PB

After eliminating the suffix redundancy (noted as ES) the storage cost reduced a lot for both methods. We achieved a significant storage reduction for all the dimensions with respect to Dwarf, because of the elimination of lexeme storage and data cube storage format in PCVDC. Again, in our system this storage was reduced by a factor of 1/N, where N is the number of column server. This is because different cuboids were stored by different servers according to the schedule tree. So it is clear that PCVDC dominate all other methods in the case of storage reduction, and suffix redundancy is a crucial factor in the overall performance.

Table 4.4: Data Cube storage cost comparison with other system (Excluding Suffix redundancy - ES)

Dimension	Dwarf ES(MB)	PCVDC ES(MB)	Storage /column server(MB)
10	62	25.26	2.53
15	153	77.38	5.16
20	300	179.65	8.98
25	516	355.09	14.20
30	812	658.69	21.95

The space required for Dwarf, PCVDC and each column server of PCVDC is plotted in the Fig. 4.3. So it is obvious that our parallel system architecture scale well with the number of dimensions. Here a constant running time can be maintained as the dimension is increased by adding a proportion number of processors.

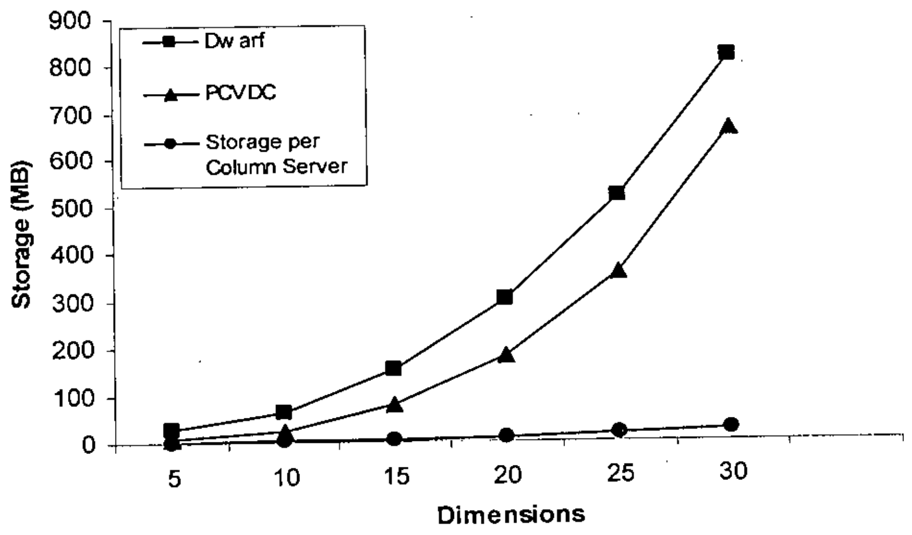


Fig. 4.3: Space requirement for Dwarf, PCVDC and each Column Server of PCVDC

4.5 Cube Construction and Query Response Time

The data cube was generated directly from this compressed vector in parallel and its contents are virtual (due to the direct absence of measure attribute), hence a great deal of reduction in the ultimate construction time was achieved. Also we speed up the processing of a query by executing in parallel the different operations in a query expression. Therefore we achieved a significant reduced query response time by applying the intraquery and interquery parallelism.

4.5.1 Cube Construction Time

We compared the construction time of PCVDC with Dwarf as shown in Fig. 4.4. The same data set was used here. Dwarf construction time was proportional to its size, and required much less time as it avoids computing large parts of the cube by eliminating the suffix redundancy. On the contrary, PCDVC creation time was little higher for smaller dimensions with respect to Dwarf. This is because, during the construction of root cuboid, it faced the data transmission overhead through the network channel. But after that all other cuboids were constructed independently in parallel from their parent cuboids in different column servers. Due to this parallel nature, a significant reduced creation time was achieved for bigger dimensions.

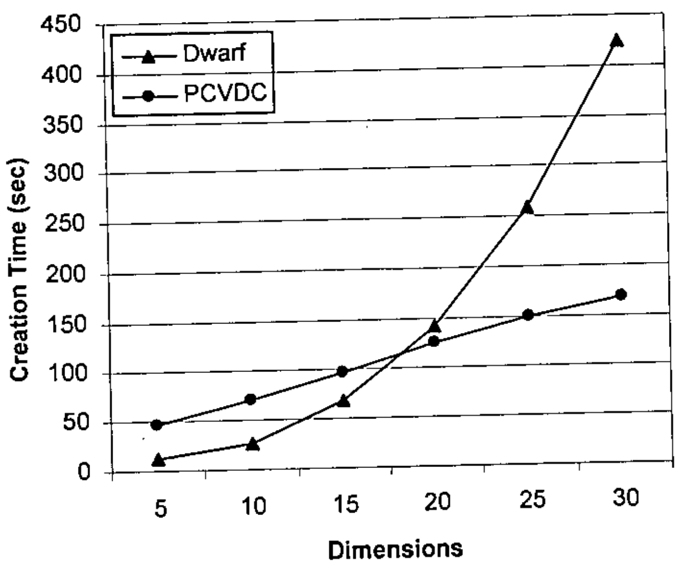


Fig. 4.4: Creation time comparison between Dwarf and PCVDC

4.5.2 Query Response Time

We first present here two types of access time performance in the query operation with respect to SQL Server: (i) Time comparison of projection operation and (ii) Time comparison of Multiple Predicates select queries. A comparison of query response time among PCVDC, Dwarf and Cubetrees [15] is also established.

4.5.2.1 Time Comparison of Projection Operation

Projection operation just accesses the specified attributes from database. To compare the access speed, we choose 4 queries shown in Table 4.5. Query Q1, Q2, Q3 and Q4 represents the projection operation of 1, 2, 3 and 4 attributes respectively. The set of quires are imposed onto 1 million tuples. Because the conventional database system accesses the entire tuple for any type of query, so the number of attributes has no significant effect in the query time. The PCVDC accesses only the information related to the attributes used in query. The average speedup for projection operation is 45.78 with respect to SQL Server.

Table 4.5: Comparison of Access Speed on Projection Queries

Sl. No.	Number of Attributes	Average Elapsed Time (sec)		Speedup (T_s/T_c)
		SQL Server (T_s)	PCVDC (T_c)	
Q1	Single attribute projection	220	6.29	34.98
Q2	Two attribute projection	285	6.85	41.60
Q3	Three attribute projection	342	7.13	47.97
Q4	Four attribute projection	457	7.80	58.59

4.5.2.2 Time Comparison of Multiple Predicates Select Queries

The Multiple Predicates SELECT queries include multiple attributes in the WHERE clause. We take here four queries given in Table 4.6. Query Q5, Q6, Q7 and Q8 has one, two, three and four attributes respectively in the WHERE clause. We run the same query in SQL Server and PCVDC. The set of quires are imposed onto 1 million tuples. The average speedup of Multiple Predicates Select Query Operation is 38.8.

Table 4.6: Multiple Predicates SELECT Queries.

Sl. No.	SQL Command	Average Elapsed Time (sec)	
		SQL Server (T_s)	PCVDC (T_c)
Q5	SELECT * FROM Sales_Table WHERE Ai='XXX' ; ($2\leq i \leq 5$)	245.43	8.24
Q6	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY'; ($2\leq i, j \leq 5$ and $i \neq j$)	337.67	9.17
Q7	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' ; ($2\leq i, j, k \leq 5$ and $i \neq j \neq k$)	388.44	9.45
Q8	SELECT * FROM Sales_Table WHERE Ai='XXX' AND Aj='YYY' AND Ak='ZZZ' AND Al='WWW' ; ($1\leq i, j, k, 2 \leq 5$ and $i \neq j \neq k \neq l$)	444.55	10.55

4.5.2.3 Query Response Time with respect to Dwarf and Cubetrees

The response time was experimented with Dwarf and Cubetrees [15] as shown in Fig. 4.5. Here the query was made against a complete cube of 4–10 dimensions with 300000 tuples in the fact table. So the number of column servers varied from 4–10. We made almost 1000 queries on projection operation and multiple predicates select queries which include multiple attributes in the where clause. Both range and point queries were made. Query against a data cube was actually be reduced to a sub-query against only one particular cuboid or a union of such sub-queries, and this is done by the query manager. The full Cubetrees [15] took a huge response time since it calculates the entire cube. Dwarf cube performed better in case of smaller dimensions but as the dimension grows it became difficult to manage the condense storage in the main memory. In contrast with that, PCVDC performed much better for higher dimensions with a little sacrifice in the query tokenization time. But it was overcome by the parallelization of the query operation. Up to 10 dimensions, the average speedup we found in the response time are 16.34 and 1.59 with respect to Cubetree [15] and Dwarf Cube respectively. This speedup would be further accelerated for higher dimensions due to our parallel architecture.

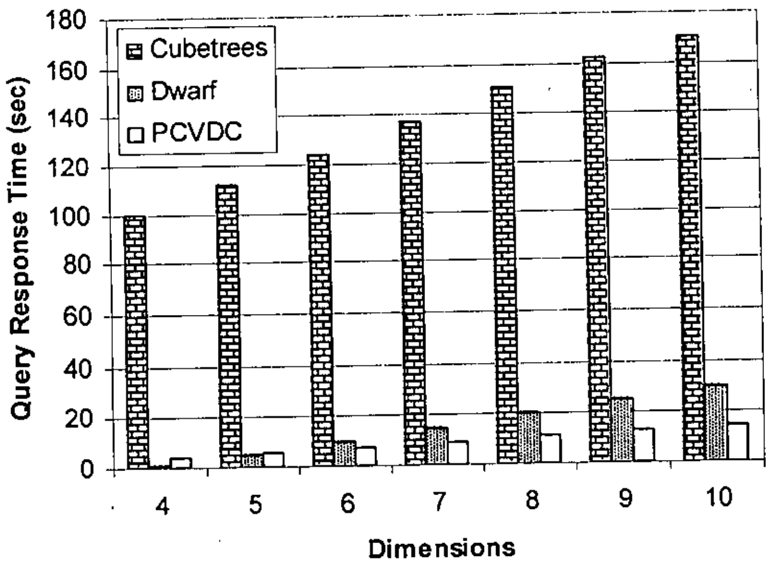


Fig.4.5: Response time of Dwarf, PCVDC and Cubetrees

4.6 Summary

In this chapter we presented the experimental evaluation of the DMBVA and PCVDC, where PCVDC is the main focus of the work. We developed a data generator which generates data that resembles the real data sets for sales relation. The simulated storage space comparison for distributed compressed data is computed using the interpolation The average compression ratio of the source relation by DMVBA is 28.87 with respect to SQL Server. Our PCVDC architecture was compared with BSF, Dwarf and Cubetrees [15]. The cube size reduction effectiveness of PCDVC was an average of 39.77% better than that of the Dwarf cube and far more against BSF. Due to this parallel nature of PCVDC, a significant reduced creation time was achieved for bigger dimensions with respect to Dwarf. In the case of query response, we got an average speedup from the virtual data cube for projection operation is 45.78 and for multiple predicates select query operation is 53.8 with respect to SQL Server. Up to 10 dimensions, the average speedup we found in the query response time are 16.34 and 1.59 with respect to Cubetree and Dwarf respectively. This speedup would be further accelerated for higher dimensions due to our parallel architecture.

Chapter 5
Conclusion and Further Research

There are works with distributed and parallel architecture of data warehouse without compression. Again, there are works with data warehouse compression but no distribution. Therefore integrating these two technologies, in this thesis, we have first presented a distributed data warehouse architecture using compression-based multi-block vector structure for relational database. Using this architecture, we achieved a compression factor around 26.45 to 31.30 against the SQL Server for source relation. Afterwards, using our PCVDC algorithm, the data cube was generated directly from this compressed form of data in a parallel environment without any decompression. Hence, it took significantly reduced creation and storage cost for the data cube. The PCVDC method was compared with BSF, Dwarf and Cubetrees. The cube size reduction effectiveness of PCDVC was around 18.88% to 59.26%, with an average of 39.77% better than that of the Dwarf cube and far more against BSF. The average speedup in the response time was 16.34 and 1.59 with respect to Cubetree and Dwarf Cube respectively.

5.1 Fundamental Contributions of the Thesis

- We have first compressed and distributed the relational data warehouse.
- In the subsequent part, we have generated a condense data cube (multidimensional view of data) directly from this compressed source relation in parallel to support fast query response.
- Suffix and prefix redundancy is eliminated from this cube to optimize the query performance. Therefore the query response time has been significantly improved.
- We made the data cube virtual in the sense that there is no direct storage of lexemes, furthermore the cube elements are stored in a compressed format.

[illegible]

- Experimental evaluation shows the improved performance over the existing systems. We have achieved a compression factor of 25-30, compared to SQL server data warehouse. Also the average data cube reduction size is a factor of 30-45, compared to existing methods.
- Moreover, the data cube is self sufficient in the sense that no need to access the fact table in answering any query and can directly extract the result using hash indexing.

5.2 Scope of Future Work

- The system has been tested using synthetic data set. It can also be implemented in real life environment. We believe that the system will behave that the system will behave as our findings.
- Data migrating tools for compression based PCVDC system in a distributed and parallel environment can be designed. Using this tool, compressed data blocks can be migrated to remote servers and query can be answered from there without any decompression.
- This architecture can be used for parallel mining, where clustering and classification techniques can be applied in parallel onto the compressed vectors and virtual data cube.
- Back-up and recovery mechanism of the distributed data cube and compressed vectors need to be considered.

References

[1] Cockshott, W. P., McGregor, D. and Wilson, J., "High-Performance Operations Using a Compressed Database Architecture", *The Computer Journal*, Vol. 41, No. 5, pp. 283-296, 1998.

[2] Hoque, A. S. M. L., "Compression of Structured and Semi-Structured Information", *Accepted for the degree of Ph. D, Dept. of Computer & Information Science*, University of Strathclyde, Glasgow, UK, 2003.

[3] Thakar, A., Szalay, A., Kunszt, P. and Gray, J., "Migrating a Multiterabyte Archive from Object to Relational Database", *IEEE Computer Society Digital Library*, Vol. 5, No. 5, pp. 16-29, 2003.

[4] Rouf, M. A. and Hoque, A. S. M. L., "Dtabase Compression for Management of Terabyte Level Relational Data", *Proceedings of 8th International Conference on Computer and Information Technology*, Dhaka, pp. 281-285, 2005.

[5] Molina, H. G., Labio, W. J., Wiener, J. L. and Zhuge, Y., "Distributed and Parallel Computing Issues in Data Warehousing", *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, Mexico, pp. 77-85, 1998.

[6] Noaman, A. Y., "Distributed Data Warehouse Architecture and Design", *Submitted for the Degree of Doctor of Pliilosophy*, University of Manitoba, Canada, 2000.

[7] Ma, H., Schewe, K. D. and Zhao, J., "Cost Optimisation for Distributed Data Warehouses", *Proceedings of the 38th Annual Hawaii International Conference on System Science,* Hawaii, pp. 283-288, 2005.

[8] Howard, P., "A New Architecture for Data Warehousing", *Bloor Research White Paper*, pp. 1-19, June 2005. http://www.itdirector.com/research.php?viewpdf=742 &mode=full.

[9] Hyperion Solution Corporation, "Large-Scale Data Warehousing Using Hyperion Essbase OLAP Technology", 1344 Crossman Avenue Sunnyvale, CA 94089, January 2000.

[10] Kalra, G. and Steiner, D., "Weather Data Warehouse: An Agent-Based Data Warehousing System," *Proceedings of the 38th Annual Hawaii International Conference on System Sciences,* Hawaii, pp. 217-221, 2005.

[11] Burns, R., Drake, D. and Winter, R., "Scaling the Enterprise Data Warehouse, Teradata's Integrated Solution," *A Winter Corporation White Paper*, Waltham, 2004, http://www.teradatalibrary.com/ pdf/teradata_7.pdf.

[12] Wiener, J. L., Gupta, H., Labio, W. J., Zhuge, Y. and Molina, H. G., "The WHIPS Prototype for Data Warehouse Creation and Maintenance," *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, Arizona, pp. 557–559, 1997.

[13] Hoque, A. S. M. L. and Siddiqui, F. H., "Parallel Virtual Data Cube Construction from a Distributed Compressed", *Proceedings of the International Conference on ICT for the Muslim World,* Malaysia, paper ID 52, pp. 1-8, 2006.

[14] Agarwal, S., Agrawal., R., Deshpande, P. M., Gupta, A., Naughton, J. F., Ramakrishnan, R. and Sarawagi, S., "On the Computation of Multidimentional Aggregates," *Proceedings of the International Conference on Very Large Databases*, pp. 506-521, 1996.

[15] Sismanis, Y., Deligiannakis, A., Roussopoulos, N. and Kotidis, Y., "Dwarf: Shrinking the PetaCube," *Proceedings of ACM SIGMOD*, Madison, USA, pp. 464-475, 2002.

[16] Feng, J., Fang, Q. and Ding, H., "PrefixCube: Prefix-Sharing Condensed Data Cube," *Proceedings of the 7th ACM International Workshop on Data Warehousing and OLAP*, Washington, DC, USA, pp. 38–47, 2004.

[17] Chen, Y., Dehne, F., Eavis, T. and Rau-Chaplin, A., "Building Large ROLAP Data Cubes in Parallel," *Proceedings of the International Database Engineering and Applications Symposium*, Montreal, pp. 367–377, 2004.

[18] Jin, R., Vaidyanathan, K., Yang, G. and Agrawal, G., "Communication and Memory Optimal Parallel Data Cube Construction", *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, pp. 1105–1119, 2005.

[19] Lawrence, R. and Kruger, A., "An Architecture for Real-Time Warehousing of Scientific Data", *Proceedings of the International Conference on Scientific Computing*, Vegus, Nevada, pp. 151-156, June 2005.

[20] Metzger, C., "The Kdb+ Database", *Kxsystems Inc.*, 2006. http:/www.kx.com.

[21] Heath, D. J., Chambers, R. W., Leggett, W. D., Tuggle, E. D., Warner, D. D., Wobschall, C. M., Fiedler, L. E. and Polluconi, M. A., "A Multi-Terabyte, Hierarchical Data Warehouse for Continuous, High-Rate Object Streams", *System and Information Review Journal*, Vol. 5, pp. 71-88, Spring 2000.

[22] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Charniack, M., Ferreira, M., Lau, E., Lin, A. and Madden, S., "C-Store: A Column-Oriented DBMS", *Proceedings of the 31st VLDB Conference*, Trondheim, Norway, pp. 553-564, September 2005.

[23] Shannon, C. E., "A Mathematical Theory of Communications", *Bell System Technical Journal*, Vol. 27, pp. 379-423, 1948.

[24] Steinmentz, R. and Nahrsteat, K., "Multimedia Computing Communication and Application", *Prentice-Hall*, Upper Saddle River, New Jersey, ISBN. No. 0-13-324435-0, pp.115-125, July 1995.

[25] Reghbati, H. K., "An Overview of the Data Compression Techniques", *IEEE Transaction on Computer*, Vol. 14, No. 4, pp. 71-75, 1981.

[26] Lee, I. and Yeom, H. Y., "A New Approach for Distributed Main Memory Database Systems: A Casual Commit Protocol", *IEICE Transaction on Inf. & Syst.*, Vol. 87-D, No. 1, pp. 196-204, 2004.

[27] Church, K. W., Fowler, G. S., Buchsbaum, A. L., Caldwell, D. F. and Muthukrishnan, S., "Engineering the Compression of Massive Tables: An Experimental Approach", *Proceedings of the 11th Annual ACM-Siam SODA*, San Francisco, pp. 175–184, January 2000

[28] Kandlur, D. D., "Computer Science Storage Systems", *Almaden Research Center*, 2005. http:/www.almaden.ibm.com

[29] Poess, M. and Potapov, D., "Data Compression in Oracle", *Proceedings of the 29th VLDB Conference*, Berlin, Germany, pp. 937-947, September 2003.

[30] Oracle Corporation, "Oracle® Database Data Warehousing Guide", ed. 10.2, 2007. http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14223.pdf.