A Personalized Smart Cube for Scalable and Faster Access to Data

by

Daniel Kwesi Antwi

Thesis submitted to the
Faculty of Graduate and Postdoctoral Studies
In partial fulfillment of the requirements
For the MCS degree in
Computer Science

School of Electrical Engineering and Computer Science
Faculty of Engineering
University of Ottawa

Abstract

Organizations own data sources that contain millions, billions or even trillions of rows and these data are usually highly dimensional in nature. Typically, these raw repositories are comprised of numerous independent data sources that are too big to be copied or joined, with the consequence that aggregations become highly problematic. Data cubes play an essential role in facilitating fast Online Analytical Processing (OLAP) in many multi-dimensional data warehouses. Current data cube computation techniques have had some success in addressing the above-mentioned aggregation problem. However, the combined problem of reducing data cube size for very large and highly dimensional databases, while guaranteeing fast query response times, has received less attention. Another issue is that most OLAP tools often causes users to be lost in the ocean of data while performing data analysis. Often, most users are interested in only a subset of the data. For example, consider in such a scenario, a business manager who wants to answer the crucial location-related business question. ”Why are my sales declining at location X”? This manager wants fast, unambiguous location-aware answers to his queries. He requires access to only the relevant filtered information, as found from the attributes that are directly correlated with his current needs. Therefore, it is important to determine and to extract, only that small data subset that is highly relevant from a particular user's location and perspective.

In this thesis, we present the Personalized Smart Cube approach to address the above-mentioned scenario. Our approach consists of two main parts. Firstly, we combine vertical partitioning, partial materialization and dynamic computation to drastically reduce the size of the computed data cube while guaranteeing fast query response times. Secondly, our personalization algorithm dynamically monitors user query pattern and creates a personalized data cube for each user. This ensures that users utilize only that small subset of data that is most relevant to them.

Our experimental evaluation of our Personalized Smart Cube approach showed that our work compared favorably with other state-of-the-art methods. We evaluated our work focusing on three main criteria, namely the storage space used, query response time and the cost savings ratio of using a personalized cube. The results showed that our algorithm materializes a relatively smaller number of views than other techniques and it also compared favourable in terms of query response time. Further, our personalization algorithm is superior to the state-of-the art Virtual Cube algorithm, when evaluated in terms of the number of user queries that were successfully answered when using a personalized cube, instead of the base cube.

Acknowledgements

I greatly thank God for helping me come this far in my education. His grace, favour and faithfulness have been unceasing in my entire life. I also express my profound appreciation to my supervisor, Dr. Herna L. Viktor for her valuable help, advice and guidance during my graduate studies and also for being pivotal in exposing to different aspects of computer science and research at the graduate level.

I sincerely acknowledge the financial support I received from the NSERC Stategic Network on Business Intelligence (BI) through my supervisors.

A special thanks goes to my parents, Mr. Patrick Antwi and Mrs. Georgina Owusua, for instilling in me, since an early age, the enthusiasm of learning and inspiring me to seek higher education and to my dear siblings, Eric Boamah, Lydia Antwi and Diana Antwi for their moral support and for always being there for me and giving me the courage to go on in my journey

Last but not least, I am grateful to my great friends, Naki Ocran for agreeing to proof-read my thesis, Mavis Manu and Anita Darkoh for always checking up on me about the status of my thesis, Michael Mireku and Dela Deyoungster for their continued support and encouragement and their insightful advice.

Contents

List of Tables

List of Figures

List of Algorithms

Chapter 1

Introduction

Data Warehouses and Online Analytical Processing (OLAP) are widely used to aid the understanding of corporate data, and aim at increasing the productivity in decision-making processes that are supported by business applications [16]. A Data Warehouse is a multidimensional database that stores subject oriented, integrated, time-variant and non-volatile data, and is often modeled through a star schema composed of fact and dimension tables [35]. While *fact tables* store numeric measures of interest, *dimension tables* contain attributes that contextualize these measures. Also, attributes of a dimension may have a relationship with other attributes of the dimension through hierarchies. A lower level to higher level hierarchy has very concise data and several intermediate levels representing increasing degrees of aggregation. All the levels together compose a data cube that allows the analyses of numerical measures from different perspectives. For example in a retail application, we can examine: (1) revenue per product, per customer city, per supplier (lower level), (2) revenue by customer country and product category, revenue per year (intermediate level), and (3) total revenue (higher level). OLAP software enables the analytical processing of a Data Warehouse's multidimensional data according to such different levels of aggregation.

In Data Warehouses, materialized views store precomputed aggregated data to eliminate overheads associated with expensive joins and/or aggregations required by analytical queries. However, due to space and time limitations, we cannot store all these views. So, given a certain storage cost threshold, there is a need for selecting the best views to be materialized, i.e. views that fit the storage requirements and provide the lowest response time to process OLAP queries. In the multidimensional model, more precisely when considering data cubes, relationships between the views may be used in order to define what is the best set of views. A view selection algorithm, in the context of data cubes takes as input a fact table and returns a set of views to store, in order to speed up queries. The so-called fact table used in the context of a data cube, is usually a flattened view that contains both numeric measures and dimension attributes. The performance of the view selection algorithm is usually measured by three criteria, namely (1) the amount of memory to store the selected views, (2) the query response time, and (3) the time complexity of the algorithm. The first two measurements deal with the output of the algorithm. Most of the works proposed in the literature consider the problem of finding the *best* data to store, in order to optimize query evaluation time, while ensuring that memory space needed by these data does not exceed a certain threshold, as fixed by the user. There are, however, some variants of the problem depending on the nature of data that can be stored, the chosen cost model and set of possible queries. The prominent role of materialized views in improving query processing performance has long been recognized. Interested readers are referred to [24][29]. Since data cubes are a set of special views, their full and partial materialization have been studied since the concept was proposed [21].

There are still pending problems with state-of-the-art OLAP techniques. That is, OLAP users are usually only interested in a subset of the data, i.e. the data that are most relevant to them. That is, on one hand they want perform analysis within this subset and on the other hand they want to view this subset as a whole cube, within which all the OLAP operators should be encapsulated. In traditional OLAP systems, in order to achieve this goal, a user must take a series of actions such as selecting the appropriate cuboid, performing SLICE or DICE operations and summing up. It takes a considerable amount of time to get the wanted information, because many steps have to be followed. Some approaches have been presented in the literature to solve this problem. Some involve updating the dimension structure, defining a constraint cube, and building a user defined virtual cube. All these can be termed as providing some form of personalization to the data cube. In the next section, we provide a motivation for our work, which also address this issue.

1.1 Motivation

In today's world, most companies increasingly collect and store business-related data into different repositories for later analysis. The need to merge such different and heterogeneous data sources has introduced an important problem for many companies; thus the problem of size of the data and dimensionality. The task to compute a data cube from such large and high dimensional data pose a problem to even the state-of-the-art approaches. The major challenges associated with data cubes computed on very large and high dimensional data are storage, cost of query processing and ability to easily get the most essential subset that is of most interest to the user. A data cube stores a large amount of redundant information. This implies that, their size quickly explode when the dimensionality increase. In the worst case, if only views are stored without indexes, the storage complexity is 2^n, where n is the number of dimensions. However, due to space and time limitations, we cannot store all these views. We therefore resort to selecting the *best* set of views that will optimize the global solution. The space consumed by a data cube is an indicator of the time it takes to create the data cube [30]. Most of the view selection solutions are designed to minimize the average query cost, but minimizing average query cost may lead to solutions where some queries are optimized while others are not. Finally, providing users with only the subset of data that is of most interest to them is crucial for productivity and key decision making. This is because data warehouses contains very large data and most of these data are of little interest to analysts. Moreover, most decision makers will prefer to have information only from the subset that is of most interest to them, with as few clicks as possible.

The problem of data cube size can be tackled by using an appropriate materialization approach. There are three main materialization approaches. The first is, full data cube materialization which stores all 2^n cuboids. This technique uses the biggest amount of space and is therefore not feasible for large and high dimensional data. The second technique is no materialization, which stores nothing and computes all queries on the-fly. However, it's query processing speed is too slow to be considered in real world applications. Finally, partial data cube materialization stores only subset of the views that will provide a good trade-off between storage space and query response time. The problem of query processing time can thus be addressed by selecting the appropriate storage structures. That is, storing only the views or storing the views together with indexes. Personalization has been the preferred approach to provide users with only the subset that is of most interest, without superfluous information.

Most of the previous studies in the literature has focused on reducing cube size and query processing time [2][42]. These proposed methods are usually based on static selection of beneficial views using some heuristic algorithm [29][30]. The problem with these approaches are that they tend to select these views in advance, based on some selection criteria, in other to optimize the global solution. However, these solutions are usually not near optimal. For example, the total benefit of the Greedy algorithm is 0.63 times the total benefit of the optimal solution [30]. Other static solutions are based on partitioning the dimension space [39]. Although data partitioning can reduce the storage space tremendously, the size of the partition might end up exploding the storage space exponentially. Other static techniques are based on using some dynamic data structures and storage techniques such as iceberg cube computation [11][28], dwarf or quotient cubes [37][61], or computation of approximate cubes [9][52]. However, the storage size of data cubes computed from these approaches explodes when dimensionality increases. An alternate to static techniques are dynamic methods, that constantly monitors user queries and materialize only those views that meet certain user criteria. Apart from cube size and query processing, there has been some studies on selecting the subset of data that is of most interest to the user [49] [65].

In a fast-changing world, users of OLAP systems require fast and unambiguous access to information. They are interested in the system making some decisions for them, based on their interest. For example, consider a scenario where a branch manager at location X wants to answer a crucial business related question "Why are sales in declining at location X". Knowing the manager's location in advance implies that the data cube can be computed such that the results of all his queries are relevant for his location. That is, we can identify that data subset that is of interest to the user and compute a personalized cube suited to that particular user. Most of the work done in the area of personalization of data cubes have focused on selecting the relevant attributes [49] based on user preference or computing virtual cubes based on user interest [65]. The problem with these approaches are that the personalized cubes are computed in advance based on interest, making the process static. User interest however, keeps changing, and therefore any method that statically select cuboids to be materialized will not be able to stand the test of time since these views will quickly become outdated.

Although the aforementioned prior research attempt to address various aspects of the problem, they do not holistically address the combined problem of storage, query processing time and the ability to provide users with only data that are of most interest to them. These techniques either solves one or two of the challenges, using either a purely static or a dynamic approach. Moreover, none of the proposed solutions is focused on drastically reducing the storage space while guaranteeing query performance. This thesis addresses these challenges, as will be discussed next.

1.2 Thesis Contribution

As mentioned previously, in this work we focus on reducing the space requirement of a data cube drastically while guaranteeing query performance by combining static and dynamic computation techniques. Secondly, we focus on providing users with only the most interesting subset of data without including superfluous information. This is also achieved by personalizing the data cube to individual users of the OLAP system. More specifically, we systematically follow a strict cube reduction strategy with the aim of reducing the size of a data cube and improving query performance. We also focus on dynamically personalizing the data cube for each user using their query patterns.

The main contribution of this thesis is in two parts. First, is the proposal of a novel algorithm, that combines *vertical partitioning, partial materialization* and *dynamic cube computation* to compute a data cube that drastically reduces the storage space while guaranteeing query response time. Secondly, we propose a dynamic personalized cube that provide users with the subset of the data that is of most interest to them.

We proceed as follows. First of all, we partitioned the dimension space vertically into disjoint sets called fragments and computed localized cube for each fragment. The computation of localized fragment cube for each fragment is done using partial cube materialization. We further introduced a new algorithm for the partial cube materialization that reduces the number of views selected for materialization while guaranteeing query performance. Our partial materialization algorithm selects only cuboids based on a performance factor. However, we eliminate cuboids that do not provide any additional benefit in terms to query performance. One problem with traditional vertical partitioning methods is that, certain queries can only be computed online by joining fragments; this is a very expensive process. In order to reduce online cube computation, we propose the Smart View algorithm. Smart Views are *top* layer views computed from attributes of different fragments, by dynamically monitoring incoming queries and materializing only the promising set of queries.

In order to ensure that users of OLAP cube are presented with only the data subset that is of most interest to them, we introduced a *personalization* algorithm, that dynamically materializes information for individual users based on their interest. Given a user's interest, a limited amount of storage space, and a filter factor, we materialize user queries that meet some pre-specified conditions. To personalize a data cube for a given user, we constantly monitor incoming queries and add user interest to the filter condition. Next, if the result query meets the filter factor and frequency threshold, it is materialized for the given user.

1.3 Thesis Organization

The remainder of this thesis consists of seven chapters. Chapter 2 provides a literature review on studies conducted in the area of data warehouse and data cubes in general. We provide a description of terms and concepts that are utilized in data warehouses and data cubes. Chapter 3 introduces data cube computation and selection techniques. We provide a description of data cube computation and present some of the computation algorithms discussed in the literature. The data cube selection process is also described. Some of the state-of-the-art data cube selection algorithms are presented and discussed. Chapter 4 is dedicated to the introduction of our algorithms. A detailed explanation of all algorithms we created is provided. Chapter 5 presents an experimental design which contains a description of the different steps involved in the algorithms we described. In Chapter 6, the criteria used to evaluate our Personalized Smart Cube approach are first presented. Our data cube algorithms and personalization algorithms are evaluated in this chapter and the results are discussed. Chapter 7 concludes and summarizes this thesis and discusses possible future work.

Part I

LITERATURE REVIEW

Chapter 2

Data Warehouses and On-line Analytical Processing (OLAP)

Data warehousing and Online Analytical Processing (OLAP) systems are valuable tools in today's competitive, fast-evolving world. Over the last several years, many organizations have spent millions of dollars in building enterprise-wide data warehouses. This is because data warehouse and OLAP are widely used to aid the understanding of organization's data, with the aim of increasing productivity in decision-making processes within the organization [16].

Recall that a data warehouse is defined as a collection of subject-oriented, integrated, non-volatile and time-variant data supporting management's decisions-making process [32]. Data warehouses generalize and consolidate data in multidimensional space. The construction of data warehouses involves data cleaning, data integration, and data transformation, and can be viewed as an important preprocessing step for data mining [27]. Moreover, data warehouses provide on-line analytical processing tools for the interactive analysis of multidimensional data of varied granularity, which facilitate effective data generalization and data mining.

OLAP has been an important tool for enterprise data analysis, as a method of organizing analysis oriented data. OLAP is a form of decision-support system that analyses data across many dimensions [57]. In OLAP applications, the data cube provides a conceptual representation of the multi-dimensional data that is generated by mapping the functional attributes of the data to the dimensions of the cube. This makes it similar to a multi-dimensional array. Multi-dimensional data cubes enables users to analyze data conveniently. As a result, the operations in OLAP that aggregate and summarize the data correspond to operations over the cells of the data cube [30]. Data cubes are defined by dimensions and facts, or measure attributes. Dimensions are the perspectives or entities with respect to which an organization wants to keep records [27]. For example, a company may create a sales data warehouse in order to keep records of the store's sales with respect to the dimensions time, item, branch, and location. The measure attributes are values of interest to an analyst, e.g. total sales, total cost, and so on.

This chapter presents an overview of the concepts of data warehouses, data warehouse architecture, and the role of OLAP data cubes within the architectural framework. We discuss the basic concept of data warehousing in Section 2.1. We first introduce the data warehouse architectures, then describe the role of the star schema for logical database design in a data warehouse. Materialized views are defined and their role in a data warehouse is also anlayzed. In Section 2.2, we concentrate on OLAP architectures and also define the concept of data cube materialization. Indexing, which is crucial for performance improvement in OLAP data cubes, is explained in Section 2.3. We study some of the most common indexing approaches used in OLAP data cubes. We finally summarize the discussion in Section 2.4.

2.1 Basic Concepts of Data Warehouse

As mentioned earlier, a data warehouse is defined as a collection of subject-oriented, integrated, non-volatile and time-variant data, supporting management's decisions-making processes [32]. These features presented distinguish data warehouse from other data repository systems, such as relational database system, transaction processing system, and file system.

- *Subject-oriented:* A data warehouse is organized around a major subject such as retail sales. Rather than concentrating on the day-to-day operations and transaction processing of an organization, a data warehouse focuses on the modeling and the analysis of data for decision makers. Hence, data warehouses typically provide a simple and concise view of particular subject issues by excluding data that are not useful in the decision support.

- *Integrated:* A data warehouse is usually constructed by integrating multiple heterogeneous sources, such as relational databases, flat files, and on-line transaction records. Data cleaning and data integration techniques are applied to ensure consistency in naming conventions, encoding structures, attribute measures and so on.

- *Time-variant:* Data are stored to provide information from a historic perspective (e.g. the last 5 - 10years). Every key structure in the data warehouse contains, either implicitly or explicitly, a time element.

- *Nonvolatile:* A data warehouse is always a physically separate store of data transformed from the application data found in the operational environment. Due to the separation, a data warehouse does not focus on transaction processing, recovery, and concurrency control mechanism. It usually focuses on providing two operations in data accessing: *initial loading of data* and *access of data*.

In summary, a data warehousing provides architectures and tools for business executives to systematically organize, understand, and use their data to make strategic decisions. A data warehouse is a semantically consistent data store that serves as a physical implementation of a decision support data model. It is often viewed as an architecture, constructed by integrating data from multiple heterogeneous sources to support structured and/or ad hoc queries, and decision making.

2.1.1 Data warehouse Architecture

In the literature, several data warehouse architectures are described [5][26], some of which include independent data marts, data mart bus architecture with linked dimensional data marts, hub-and-spoke, centralized data warehouse(no dependent data marts), and federated. All these can be classified as adopting a three tier architecture [27]. Figure 2.1 shows a simplified three (3) tier architecture.

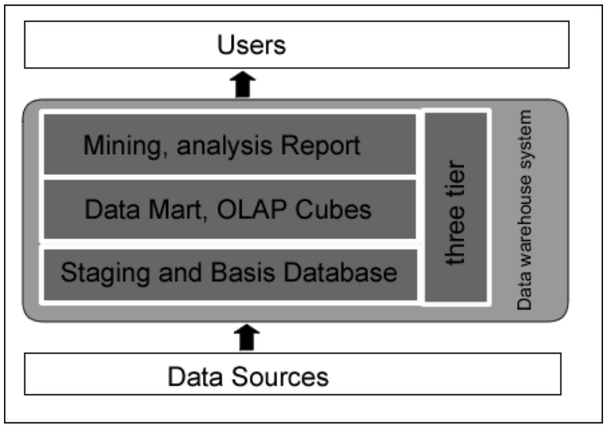


Figure 2.1: A sample three (3) tier data warehouse architecture

The bottom tier is a warehouse database server that is in most cases, a relational system. However, a multidimensional system is some times used. Most relational databases used as a warehouse database are modeled using the star schema, since it minimizes the joins required for query computation [35]. Back-end tools and utilities are used to feed data into the bottom tier from operations databases or other external sources (e.g., customer profile information provided by external consultants). These tools and utilities perform extraction, cleaning and transformation (e.g., to merge similar data from different sources into a unified format), as well as load and refresh functions to update the data warehouse. The data are extracted using application program interfaces known as gateways. A gateway e.g. Object Linking and Embedding Database (OLEDB) is supported by the underlining Database Management System (DBMS) and allows clients programs to generate SQL or MDX code to be executed at a server. This tier also stores the meta-data repository. The middle tier is made up of the OLAP Server (data cubes) and data marts. Data cubes are built based on the data structure and granularity of data in the bottom tie. These data cubes are redundant copies of data that are defined according to the users requirements for analysis. OLAP servers are implemented using either a relational OLAP (ROLAP) model or a multidimensional OLAP (MOLAP) model. The top tier is the front-end client layer, which contains query and reporting tools, analysis tools, and/or data mining tools.

2.1.2 Star Schema - A logical database design

Recall that the star schema design is often used to model a data warehouse database. This database consists of a *fact table* that describes all transactions and a *dimension table* for each entry [14]. For example, in a fictitious data warehouse, each sales transaction involves several entries - a customer, a salesperson, a product, an order, a transaction date, and the city where the transaction occurred. Each entity also has measure attributes - the number of units sold and the total amount the customer paid. Each tuple in the fact table consists of a reference, by means of a foreign key, to each entity in a transaction and the numeric measures associated with the transaction. Each dimension table consists of columns that correspond to the entry's attributes. Computing the join between the fact table and a set of dimension tables is more efficient than computing a join among arbitrary relations.

Some entities, however, are associated with hierarchies, which the star schema do not explicitly support. A hierarchy is a multilevel grouping in which each level consists of a disjoint grouping of the values in the level immediately below it. For example, all products can be grouped into disjoint sets of categories, which are themselves grouped into disjoint set of families.

2.1.3 Materialized Views

Many data warehouse queries require summary data and therefore use aggregates. Materializing summary data can accelerate common queries [14] and eliminate overheads associated to expensive joins or aggregations required by analytical queries. However, due to space or time limitations, we cannot store the result of all queries. So, one has to select the *best* set of queries to materialize. Thus the objective is to select the set of views that can efficiently answer as many queries as possible. In the multidimensional model, more precisely when considering data cubes, relationships between the views can be used to order to define what is the best set of views. Recall, a view selection algorithm in the context of data cubes takes as input a fact table and returns a set of views to store, in order to speed up queries. The selection of part of the data cube for materialization is a multi-criteria task [42]. Some of these are, the materialization granularity (full vs fragments of views), the constraints taken into account (available storage space and/or time window allowed for incremental update), presence of a target workload query, complexity of the view selection algorithm, dynamic workload and the presence or absence of indexes. As mentioned earlier, the performance of the view selection algorithms is usually measured by three criteria [29], namely (1) the amount of memory to store the selected views, (2) the query response time and (3) the time complexity of this algorithm. Most of the proposed solutions formalizes the view selection problem so that the returned solution minimizes the average query cost, while satisfying the imposed budget space and/or update time constraints.

2.2 Data Cubes and OLAP

Recall that data warehouse and OLAP tools are based on a multidimensional data model [27]. This model views data in the form of a data cube. These systems help to analyze complex multidimensional data and provide decision support [60].

2.2.1 Data Cube: A Multidimensional Data Model

Data cubes are powerful tools that allow data to be modeled and viewed in multiple dimensions. They support the analysis of the contents of data warehouses and databases [50]. A data cube consists of the results of group-by aggregate queries on all possible combinations of the dimension attributes over a fact table in a data warehouse. The group-by construct is used by SQL to create a table of many aggregate values indexed by a set of attributes. Materialization of summary views on the cube is critical for improving the response time of OLAP queries and of operators such as roll-up, drill-down, and pivot A data cube can be referred to as a multidimensional array. It is defined by dimension attributes and measure attributes. In general terms, dimensions are the perspectives or entities with respect to which an organization wants to keep records. Dimension attributes are sometimes referred to as functional attributes [19][50]. Each dimension may have a table associated with it, called a dimension table, which further describes the dimension. A multidimensional data model is typically organized around a central theme, such as sales. This theme is represented by measure attributes. Recall that, these are attributes whose values are of interest to the analyst. The measures are contained in a table called the fact table. A cell of a data cube is described by a unique combination of dimension values.

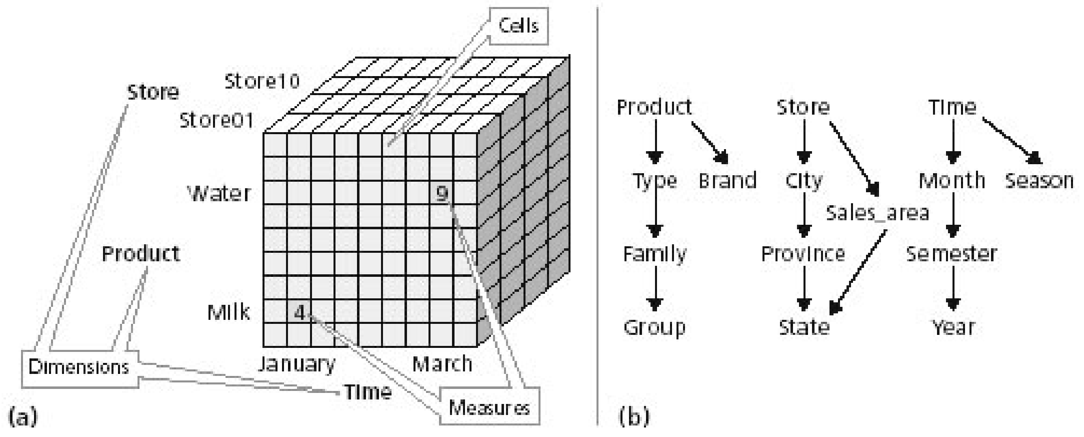
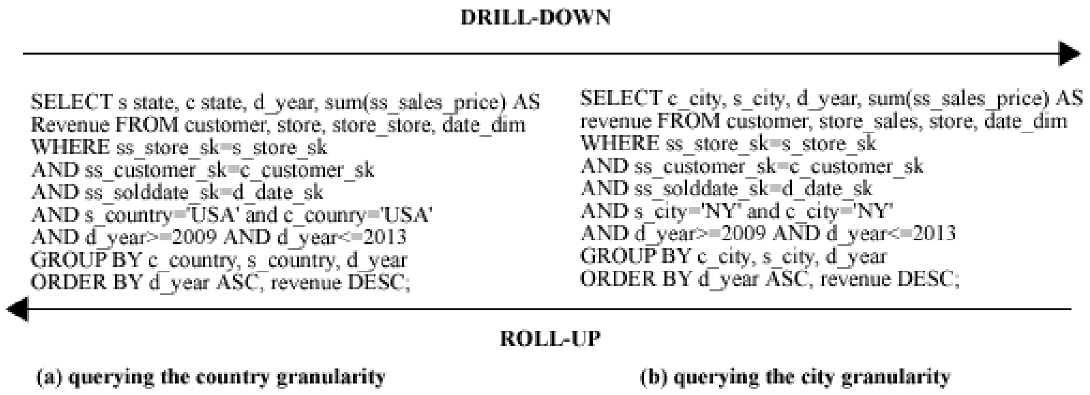


Figure 2.2: *A multidimensional model data cube* [33]

Figure 2.2 depicts a small, practical data cube example, that considers a hypothetical database of sales information maintained by a company. This particular data cube has three dimension attributes - store, product, and time - and a single measure attribute - product sales for a large chain of stores (sales is computed with the sum function). By selecting cells, planes, or sub cubes from the base cuboid, we can analyze sales figures at varying granularity. This example data cube can provide an aggregated total orders for all combinations of stores, product and time. Such queries form the basis of OLAP functions like roll-up and drill down.

Drill-down and *roll-up* OLAP operations depend on hierarchies [62] as introduced earlier. A *drill-down* operation decomposes fact table data to lower levels of a hierarchy, then increasing data details. Inversely, a *roll-up* operation aggregates fact table data to upper levels of a hierarchy, then summarizing the data. Figure 2.3 shows example of these operations adapted from [45], using existing hierarchies held by the *Customer* and *Store* dimension tables. Notice the use of the group-by construct in this SQL statements. Considering that the user firstly issued the query of Figure 2.3a and later issued the query of Figure 2.3b, there was a *drill-down* operation based on both $(c\_country) \preceq (c\_city)$ and $(s\_country) \preceq (s\_city)$. On the other hand, if the user had issued the query inversely, there was *roll-up* operation based also on those mentioned hierarchies. The underlined attributes in Figure 2.3 highlights these operations



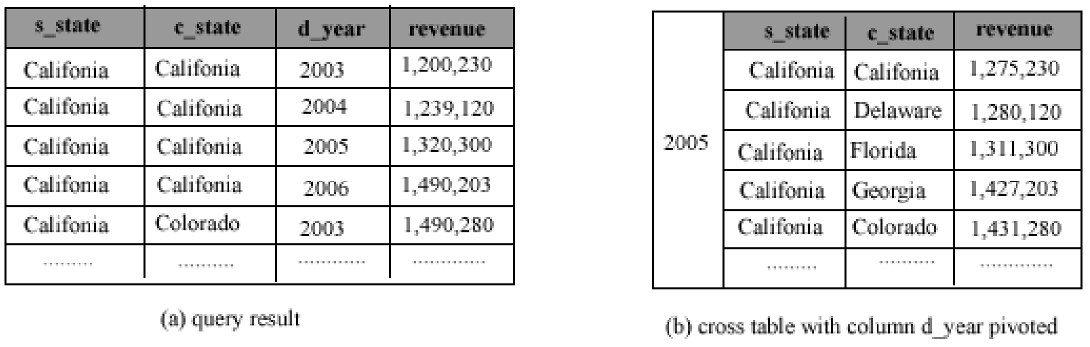
```
SELECT s state, c state, d_year, sum(ss_sales_price) AS
Revenue FROM customer, store, store_store, date_dim
WHERE ss_store_sk=s_store_sk
AND ss_customer_sk=c_customer_sk
AND ss_solddate_sk=d_date_sk
AND s_country='USA' and c_counry='USA'
AND d_year>=2009 AND d_year<=2013
GROUP BY c_country, s_country, d_year
ORDER BY d_year ASC, revenue DESC;
```

(a) querying the country granularity

```
SELECT c_city, s_city, d_year, sum(ss_sales_price) AS
revenue FROM customer, store_sales, store, date_dim
WHERE ss_store_sk=s_store_sk
AND ss_customer_sk=c_customer_sk
AND ss_solddate_sk=d_date_sk
AND s_city='NY' and c_city='NY'
AND d_year>=2009 AND d_year<=2013
GROUP BY c_city, s_city, d_year
ORDER BY d_year ASC, revenue DESC;
```

(b) querying the city granularity

Figure 2.3: Roll-up and drill-down operations



s_state	c_state	d_year	revenue
Califonia	Califonia	2003	1,200,230
Califonia	Califonia	2004	1,239,120
Califonia	Califonia	2005	1,320,300
Califonia	Califonia	2006	1,490,203
Califonia	Colorado	2003	1,490,280
....	....	....	....

(a) query result

	s_state	c_state	revenue
2005	Califonia	Califonia	1,275,230
	Califonia	Delaware	1,280,120
	Califonia	Florida	1,311,300
	Califonia	Georgia	1,427,203
	Califonia	Colorado	1,431,280
	....	....	....

(b) cross table with column d_year pivoted

Figure 2.4: The original query results and the pivoted query results

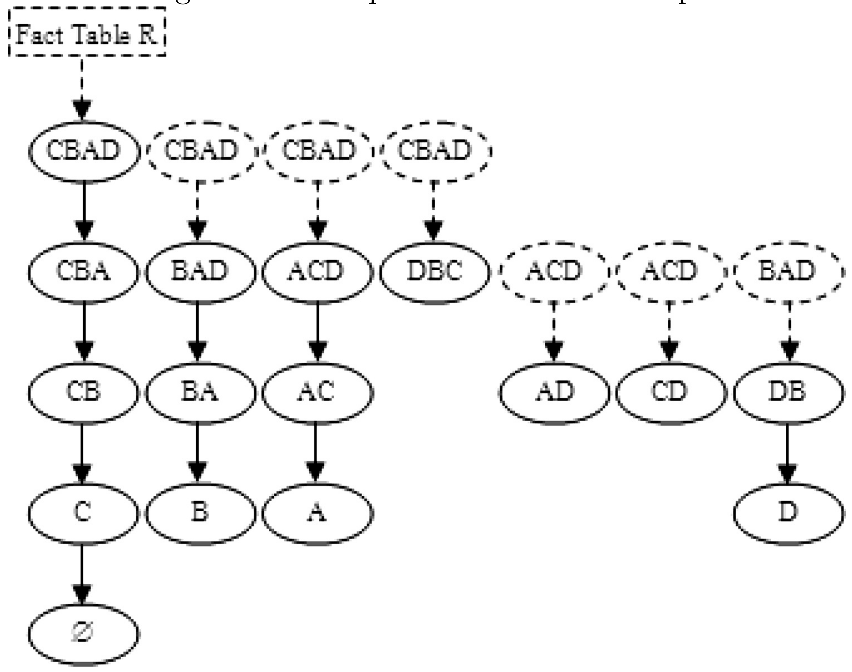
Both the queries of Figure 2.3a and 2.3b exemplify the *slice and dice* operation, which consists of applying filters to the resulting data, such as *"c_country='USA' AND s_country='USA' AND d_year* >= 2003 *AND d_year* <= 2013*"*, shown in Figure 2.3a. Finally, the *pivoting* operation enables reordering results by switching the axis for columns and rows [15]. Figure 2.4 shows the results of the query in Figure 2.3a, whose column d_year was pivoted to be a row, providing the results of Figure 2.4b. The representation of results in Figure 2.4b is also known as a *cross table.*

In total, a d-dimensional base cube is associated with 2^d cuboids. Each cuboid represents a unique view of the data at a given level of granularity. However, storing all 2^d cuboids is not a feasible approach, since it becomes very large with a large d. Selecting the appropriate cuboids for materialization is therefore crucial for an efficient data cube.

2.2.2 Data cube traversal

Data cubes can be traversed in a top-down fashion, for example Pipesort [2] or bottom-up fashion, for example BottomUp Cube [11]. PipeSort follows paths in the search lattice of the data cube. In the example in Figure 2.5, the raw data may be sorted first in a first attribute order, such as C - B - A - D. Having sorted the data, cuboids sharing some parts of the sort order may be evaluated. These cuboids are said to have a "prefix" common to the sort order. In this example, (C, B, A, D), (C, B, A), (C, B) and C will be an ideal traversal order. This order may also incorporate the smallest-parent objective of Gray et al. [21].

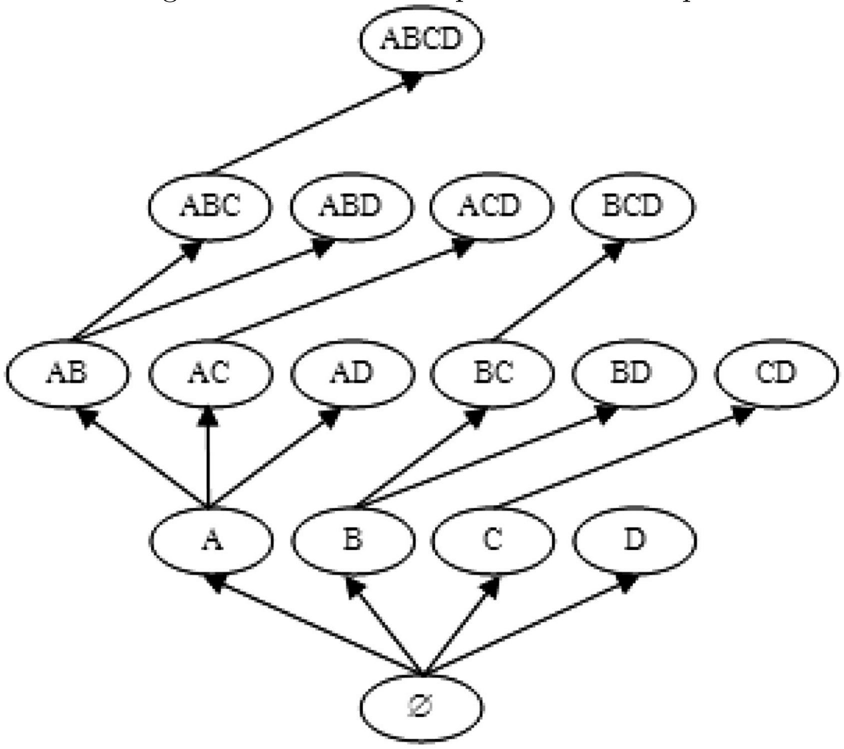
Figure 2.5: A top-down traversal example



BUC on the other hand traverses the data cube in a bottom-up and depth-first fashion. Lets consider the example shown in Figure 2.6. This approach start with the *ALL* node, which contains only one tuple, moving towards more and more detailed nodes with more grouping attributes. Then, it sorts the relation according to A and isolates the first set of tuples S_{A1} that share the same value in A. It proceeds recursively to node

AB passing S_{A1} as input. In that call, it re-sorts S_{A1} according to B and isolates the first set of tuples S_{A1B1} that share the same values. It continues recursively finishing at the root of the lattice, which contains all dimensions as its grouping attributes.

Figure 2.6: A bottom up traversal example



2.2.3 Data Cube Materialization

Recall that many data warehouse queries require summary data and therefore use aggregates. Materialized views have been recognized as an effective query optimization technique for a long time [13]. Since data cubes [21] are sets of special views, their full and partial materialization have been studied as soon as this concept was proposed. The challenges in exploiting materialized views are similar to the indexing challenges. These are, identifying the best set of views to materialize [23][30], exploiting the materialized views to answer queries, and updating the materialized views during load and refresh. The three main choices for data materialization given a base cuboid are *No materialization*, *Full materialization* and *Partial materialization*.

No Materialization

This does not precompute any of the "nonbase" cubiods and thus leads to computing expensive multidimensional aggregates "on-the-fly',' from raw data on request. No materialization has the problem of basing its performance on quick query response from the database system where the raw data is stored. No extra space beyond that for the raw data is required.

Full Materialization

Full materialization precomputes all of the cuboids. The resulting lattice of computed cuboids is referred to as the *full cube.* This approach gives the best query response time. However, precomputing and storing every cell is not a feasible alternative for most data cubes, as the space consumed becomes excessive. This is because of the fact that, for a very large database with high dimensionality, precomputing all 2^d cuboids, where d is the number of dimensions, it is prohibitive in terms of storage requirement and computation time. It should be noted that the space consumed by the data cube is also a good indicator of the time it takes to create the data cube, which is important in many applications. If the data cube is to be indexed, then the spaced consumed also impacts indexing and thus adds to the overall cost.

Partial Materialization

Partial materialization selectively computes a proper subset of the whole set of possible cuboids. Partial materialization may also involve computing a subset of the cube, which contains only those cells that satisfy some user-specified criterion, such as where tuple count of each cell is above some threshold [11]. This approach is made possible because, in data cubes, the values of many cells are computable from those of other cells in the data cube [21][29]. This dependency is similar to spreadsheets, where the value of cells can be expressed as a function of the values of other cells. We call these cells "dependent" cells. For example given (A,B,C) as a *3D* data cube, we can compute the values of cells (A,ALL,C) as the sum of the values of cells of (A,B_1,C),.....,(A,B_n,C), where B_n is the number of items of B. The more cells that are materialized, the better the query response time. For large data cubes, however, we may be able to materialize only a small fraction of the cells of the data cube, due to space and other constraints. Partial materialization represents an interesting trade-off between storage space and response time [30]; thus its important to pick the right cells to materialize. Partial Materialization of cuboids should consider three factors. These are, identifying the subset of cuboids or sub-cubes to materialize, exploiting the materialized cuboids or sub-cubes during query processing and efficiently updating the materialized cuboids during load and refresh. Although selecting the right materialization algorithm is crucial for the performance of an OLAP data cube, its also important to select the right OLAP data cube architecture.

2.2.4 OLAP Architecture

Logically, OLAP servers present business users with multidimensional data from data warehouses or data marts, without concerns regarding how or where the data are stored. However, the physical architecture and implementation of OLAP servers must consider data storage issues [27]. There are three main architecture implementations [14][27], as will be discussed next.

Relational OLAP (ROLAP)

Relational OLAP servers are the middleware servers that sit in-between the relational back-end server where the data warehouse is stored and the client front-end tools. ROLAP servers support multidimensional OLAP queries. It includes optimization for each database management system (DBMS) back end, implementation of aggregation navigation logic, and additional tools and services. ROLAP servers identify the views to be materialized, rephrase user queries in terms of the appropriate materialized views, and generate multi statement SQL for the back-end server. A noted advantage over the other architectures is that ROLAP technology tends to have greater scalability. Also relational data can be stored more efficiently than multidimensional data [59], although improvements to MOLAP architecture is making the difference less obvious. ROLAP also provides additional services such as query scheduling and resource assignment. Although these servers exploit the scalability and transactional features of relational systems, intrinsic mismatches between OLAP-style querying and SQL can create performance bottlenecks in OLAP servers. Bottlenecks are, however, becoming less of a problem with OLAP-specific SQL extensions implemented in relational servers such as Oracle, IBM DB2, and Microsoft SQL Server. The addition of functions such as median, mode, rank, and percentile has extended the aggregate functions which are a key part of most data cube computations. Other feature additions include aggregate computation over moving windows, running totals, and breakpoints to enhance support for reporting applications.

A major problem with ROLAP was that multidimensional spreadsheets require grouping by different sets of attributes. In order to solve this problem Gray et al [21] proposed two operators - *roll-up* and *cube* to augment SQL and address this requirement. Roll-up of a list of attributes such as product, year, and city over a data set results in answer sets with the following applications; (a) a query result that is grouped by products, year, and city; (b) a query result that is grouped by product and year; and (c) a query result that is grouped by product. Given a list of k columns, the cube operator provides a group-by for each of the 2^k combinations of columns. Such multiple group-by operations can be executed efficiently by recognizing commonalities among them. When applicable, precomputing can enhance OLAP server performance.

Multidimensional OLAP (MOLAP)

MOLAP is a server architecture that does not exploit the functionality of a relational back end, but directly supports a multidimensional data view through array-based multidimensional storage engines [14]. MOLAP map multidimensional views directly to data cube array structures. These array structures are basically n-dimensional array data structures used to store the data cubes. They also enable implementation of multidimensional queries on storage layer view direct mapping. One principal advantage of MOLAP over ROLAP is it's excellent indexing properties; its disadvantage is poor storage utilization, especially when the data is sparse [51]. However, sparse matrix compression techniques may be explored. Many MOLAP servers adapt to sparse data sets through a two-level storage representation and extensive compression. In a two-level storage representation, denser sub-cubes are identified and stored as array structures, whereas sparse sub-cubes employ compression technology for efficient storage utilization. Traditional indexing structures can be used to index these dense smaller arrays. For example, consider a fictitious database with 11 dimension attributes *age, marital_status, gender, eduction, race, origin, family_type, detailed_household_summary, age_group*, and *class_of_worker* and measure attribute *income*. This data cube contains more than 16 million cells with only 16000 nonzero elements. In such a data cube the density is 0.001. In reality, many data warehouses contain multiple small regions of a clustered dense regions, with points sparsely scattered over the rest of the space [38][66]. Although MOLAP servers offer good performance and functionality, they do not scale well for extremely large data sizes [14].

Hybrid OLAP (HOLAP)

The hybrid OLAP approach combines ROLAP and MOLAP technologies, benefiting from the greater scalability of ROLAP and the faster computation of MOLAP [14]. HOLAP servers identify sparse and dense regions of the multidimensional space and take the ROLAP approach for sparse regions and MOLAP approach for dense regions. HOLAP servers split a query into multiple queries, issues the queries against the relevant data portions, combines the results, and then presents the result to the user. HOLAPs selective view materialization, selective index building, and query and resource scheduling are similar to its MOLAP and ROLAP counterparts. Selecting the right architecture based on the nature of data is important for building efficient OLAP data cube. However, performance of an OLAP data cube, irrespective of the selected architecture, is determined by the indexes selected and may be difficult to tune.

2.3 Indexing Data Cubes

Queries defined on Data Warehouses (called star join queries) are complex, since they involve several joins, group-bys and selections. Indexes are solid candidates to optimize such operations and considered as the foundation of the optimization techniques in databases and data warehousing [10]. Indexing represent an important part of any database system design as they can significantly impact workload performance, by enabling quicker and more efficient access to data. In the Data Warehouse context, when we talk about indexing, we refer to two different aspects, namely (i) indexing techniques and (ii) the index selection problem.

2.3.1 Bitmap Indexes

A promising approach to process complex queries in OLAP environment is to use bitmap indexes [46][47]. A Bitmap index on an indexed attribute consists of one vector of bits (i.e., bitmap) per attribute value, where the size of each bitmap is equal to the cardinality of the indexed relation. The bitmaps are encoded such that the i^{th} record has a value of v in the indexed attribute if and only if the i^{th} bit in each of the other bitmaps is set to 0. This is called a Value-List index [46]. Consider a high-selectivity-factor query with selection predicates on two different attributes. A conventional database optimizer would generate one of the following three query plans: (P1) a full relation scan, (P2) an index scan (using the predicate with lower selectivity factor) followed by a partial relation scan to filter out the non-qualifying tuples, or (P3) an index scan for each selection predicate, followed by a merge of the results from the two index scans. Due to the compact sizes of bitmaps (especially for attributes with low cardinality) and the efficient hardware support for bitmap operations (AND, OR, XOR, NOT), plan (P3) using bitmap indexes is likely to be more efficient than a plan that requires a partial or full relation scan (plans (P1) and (P2)). A simple cost analysis shows that evaluating plan (P3) with bitmap indexes is often more efficient than using the conventional tid-list (tuple identifier list) based indexes for queries with selectivity factor above some attribute cardinality threshold.

This is because tid-list based index also called inverted index store the actual values of the record identifiers instead of 0s and 1s for the bitmap index, and is therefore not very efficient for low cardinality attributes. Let N and r be the relation and query result cardinalities, respectively. Assume that each tid is 4 bytes long and that one bitmap is scanned per predicate. In terms of the number of bytes read, using bitmap indexes for plan (P3) is more efficient than using tid-list based index if $2\frac{N}{8} \leq 4(2r)$; i.e., $\frac{r}{N} \geq \frac{1}{32}$. Further operations on bitmaps are more CPU-efficient than merging tid-lists.

2.3.2 Bitmap Join Indexes

A Join index, considered as multiple table index, is well adapted for queries that require fetching information from multiple tables. It involves calculating the result of joining two tables on a join attribute and projecting the keys of the two tables. To join two tables, one may thus use the join index to fetch the tuples from the tables followed by a join. In a Data Warehouse, it is of interest to perform multiple joins on a fact table and the corresponding dimension tables. Therefore, building a join index between the keys of the dimension tables and the corresponding foreign keys of the fact table is very important. If the join indexes are represented in bitmap, it is called a Bitmap Join Index [10]. Using a Bitmap Join index, a multiple join could be replaced by a sequence of bitwise operations, followed by a relatively small number of fetch and join operations.

2.3.3 Inverted Indexes

Inverted indexes are well-known methods in information retrieval [7]. An inverted index for a collection of documents is a structure that stores, for each term (word) occurring somewhere in the collection, information about the locations where it occurs. In particular, for each term t, the index contains an inverted list, consisting of a number of index postings. Each posting in the index contains information about the occurrences of t in one particular document d, usually the *id* of the document (the docID), the number of occurrences of t in d (the frequency), and possibly other information about the locations of the occurrences within the document and their contexts [64]. For example, an inverted index list I_t of the form $\{56, 1{,}34\}$ $\{198, 2{,}14{,}23\}$ might indicate that term t occurs once in document 56, at word position 34 from the beginning of the document, and twice in document 198 at positions 14 and 23. We assume postings have docIDs and frequencies but do not consider other data such as positions or contexts.

An inverted index for an attribute in a Data Warehouse consists of a dictionary of the distinct values in the attribute, with pointers to inverted lists that reference tuples with

Table 2.1: The original table from a sample database

tid	A	B	C
1	a1	b1	c1
2	a1	b2	c1
3	a1	b2	c1
4	a2	b1	c1
5	a2	b1	c1

the given value through tuple identifiers (tids) [12]. In its simplest form it is constructed as follows. For each attribute value in each dimension, we register a list of tuple id (*tid*) or record id (*rid*) associated with it. For example, in Table 2.1, attribute value a2 appears in tuples 4 and 5. The tid-list for a2 as shown in Table 2.2, contains exactly 2 items, namely 4 and 5. The resulting inverted index for the 5 individual dimensions are shown in Table 2.2. The inverted index for multiple dimensions can also be computed by using set operations of Intersection and Union. For example, Table 3.4 show the result of a two dimensional cuboid AB from Table 3.4. This uses set intersection to compute the intersection of the tid-lists of dimension A, and the tid-list of dimension B. The computation of the two dimensional cuboid from a one dimensional cuboid is done in a bottom-up fashion, similar to the Apriori algorithm for finding frequent item-sets [3]. To reduce both space usage and I/O requirements in query processing, the inverted lists are often compressed by storing the deltas (or offsets) between the stored references [67]. The deltas are bit codes for representing very large numbers in other to reduce its size. This approach makes small values more likely, and several compression schemes that represent small values compactly have been suggested. One of the most efficient compression methods according to recent study [12] is the PForDelta technique [68]. PForDelta stores deltas in a word-aligned version of bit packing, which also includes exceptions to enable storing larger values than the chosen number of bits allowed.

2.4 Summary

In this chapter, we have outlined some descriptions and concepts relevant to OLAP data cubes and data warehousing. We presented the concepts of data warehousing, explained the basic architecture of a data warehouse and described the concept of materialized views required for efficient cube computation. We also studied the concept of OLAP and data cubes where we explained that OLAP systems are based on models that views data in the form of a data cube. The different OLAP architectures were presented and

Table 2.2: Inverted Index for a single dimension A

Attribute Value	TID List	List Size
a1	1 2 3	3
a2	4 5	2
b1	1 4 5	3
b2	2 3	2
c1	1 2 3 4 5	5
d1	1 3 4 5	4
d2	2	1
e1	1 2	2
e2	3 4	2

Table 2.3: Cuboid AB using inverted index

Cell	Intersection	Tuple ID List	List Size
a1 b1	1 2 3 $\cap$1 4 5	1	1
a1 b2	1 2 3 $\cap$2 3	2 3	2
a2 b2	4 5 $\cap$2 3	ϕ	0

explained. MOLAP servers supports a multidimensional data view through array-based multidimensional storage engine. ROLAP servers sit in-between the relational back-end server where the data warehouse is stored and the client front-end tools, and generate and translate queries based on appropriate materialized views. Finally, Hybrid OLAP counter the disadvantages of the two architecture approaches. Data cube materialization is an essential part of data cube construction, since it impacts both storage requirement and query processing. The materialization option determines the space and time complexity of the data cube. First, No Materialization has a very low space complexity since only the source fact table is stored and very high time complexity. Full Materialization implies very high space complexity, since the entire cube is stored and very low time complexity. Partial materialization gives a trade-off between space and time complexity. Finally, we introduced some indexing techniques used in OLAP data cubes, since it is crucial for query performance improvement. We showed how indexing may help improve query response time by using small and simple data structures or by reducing the space required to store data using bits to represent data attribute values.

The next chapter introduces data cube techniques that are relevant for the Personalized Smart Cube approach developed in this thesis.

Chapter 3

Data Cube Computation Techniques

Recall that data cube implementation is one of the most important and expensive process in online analytical processing. It involves computing and storing of the results of aggregate queries groupings on dimensions-attribute combinations over a fact table in a data warehouse [42]. The precomputation and materialization of parts of, or the whole of, the data cube is fundamental for improving the query response time of OLAP queries and of operators such as roll-up, drill-down, slice-and-dice, and pivot [15]. Full data materialization is ideal for fast access to stored cuboids, but will pose considerable costs both in computation time and in storage space. In order to balance this tradeoff between query response time and data cube storage requirements, several methods have been proposed in the literature. We address the question: "How can we compute data cubes in advance, so that they are handy and readily available for query processing?". As a running example, we will consider a data cube that illustrates the computational dependencies among the different group-bys in the cube lattice [30]. Figure 3.1 presents cube lattice of a fact table R with three dimensions (A, B, C), where a lattice node label is a concatenation of the corresponding dimension names.

In this chapter we study data cube computation, data cube selection and various cube materialization (i.e., precomputation) strategies. We initially explain data cube computation methods and introduce existing algorithms in Section 3.1. We then study data cube selection methods in Section 3.2. Finally, in Section 3.3 we summarize this chapter.

3.1 Computation Methods

Data cube computation involves scanning the original data, applying the required aggregate function on all groupings, and generating the cube contents [42]. The main objective of computation algorithms is to place tuples that aggregate together (i.e, tuples with the same values in the grouping attributes) in adjacent in-memory positions, so that all group-bys can be computed with as few data scans as possible. The two main alternative approaches that may be used to achieve such a tuple placement are sorting and hashing [21] [33]. These are used to organize the data by value and then aggregate with a sequential scan of the sorted data. As shown in 3.1, the lattice structure indicates there is much commonality between a parent node and its children. Taking advantage of this commonality may lead to particularly efficient algorithms. For example, if the initial data in Figure 3.1 is sorted according to attributes ABC, then by sharing the sorting cost, nodes $ABC \rightarrow AB \rightarrow A \rightarrow \phi$ can be computed in a pipeline fashion. Assumming A is *country*, B is *city* and C is *store* then, the cuboid ABC can be computed using the group-by statement *SELECT Country, City, Store FROM Tablename GROUP BY Country, City, Store* and AC can be computed by *SELECT Country, City FROM tablename GROUP BY Country, City* where *tablename* is the name of the fact table. Sorting and hashing are most efficient when the data processed fits in memory; otherwise, external memory algorithms must be used, which incur greater I/O cost. To overcome this drawback, most computation methods partition data into disjoint fragments that do fit in memory, called partitions. Tuples are not placed into partitions randomly, but according to the policy that tuples that aggregate together must belong to the same partition.

3.1.1 2^D Algorithm

The 2^D algorithm [21] is a simple, initial algorithm for data cube computation [42]. It computes each group-by directly from the original fact table and then takes the union of all partial results for the data cube. The 2^D algorithm is not efficient, since it has exponential complexity with respect to the number of dimensions and takes no advantage of the commonalities among the interim results of different group-bys. Hence, any data structures, tuple sorting, or hash tables constructed for the computation of one group-bys are never reused but are recreated from scratch, whenever necessary. Such behavior renders it impractical. To overcome the main problems of this algorithm, all subsequent algorithms identify a spanning tree T of the lattice whose edges indicate which node (parent) will be used for computation of each node (child).

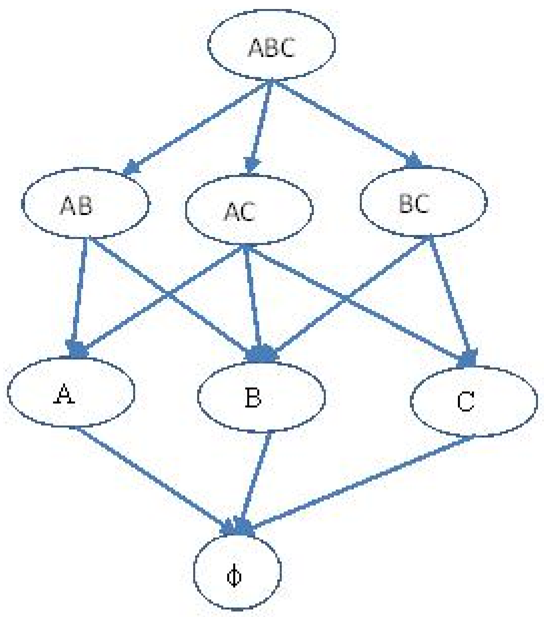


Figure 3.1: Example of cube lattice

3.1.2 GBLP Algorithm

The GBLP algorithm [21] improves the efficiency of the 2^D algorithm by computing each node in the lattice using its smallest parent and not the original fact table. The group-by operation may either be sort-based or hash-based, depending on the implementation. The size of nodes tends to decrease as groups of tuples get aggregated and are replaced with less detailed summary tuples as we move towards lower levels in the lattice. This implies that selecting the smallest parent node always leads to faster computation of a node. This algorithm computes each child node from the parent with the smallest size. In reality, this algorithm obtains estimates of the size of each lattice node based on a variety of statistical methods such as those presented in [25], [51], [54]. The main disadvantage of the GBLP algorithm is that it only prunes the lattice into a tree, without suggesting a particular way to traverse it. Lack of an efficient disk-access plan makes GBLP impractical when dealing with large datasets that do not fit into main memory. Gray et al, in their paper, presented a rough comparison between the size of the fact table and the corresponding data cube. The size C of the fact table (in terms of number of tuples) is approximated by $C = C_1 \times C_2 \times C_3 \cdots \times C_D$, where C_i denotes the cardinality (domain size) of the i-th dimension (i=1...D). The data cube only adds the value (ALL) to each dimension, so the size in terms of number of tuples, is approximated by $C_{cube} =$

$(C_1+1)\times(C_2+1)\times\cdots\times(C_D+1)$, making it slightly larger than the original fact table. An argument is therefore made that, whenever the original fact table fits in memory, it is highly likely that the data cube also fits in memory. The strength of this argument is weakened [30] by the fact that C and C_{cube}, are actually rough estimations of the sizes of the fact table and the data cube, respectively. In most real-world applications, the corresponding formulas do not hold. An original fact table is generally sparse, so this implies that its actual size is a small fraction of the size of the Cartesian product of the domains of its attributes. The sparser the fact table is, the higher the quotient of the cube size over the fact-table size is. Thus, the size of the data cube is expected to be much larger than the size of the fact table in terms of number of tuples. To overcome these memory size issues, other algorithms partition their execution tree into an appropriate set of subtrees, each one of which has a small enough number of nodes so that they can all be constructed concurrently, reducing memory requirements overall.

3.1.3 Partitioned-Cube Algorithm

Data in real applications tends to be sparse due to the following two reasons [51]. Firstly, the number of dimensions of the original fact table is large and secondly some dimensions of the original fact table have large domains. Recall that a data cube is sparse when a large percentage of data cells contain zeros (0). Sparsity results in fewer aggregation operations, since in sparse datasets, the number of tuples with the same dimension values decreases. Consequently, the relative sizes of intermediate nodes in the cube lattice tend to increase with sparsity. A large sizes of intermediate nodes introduce considerable I/O costs to the computation methods presented here, since these methods sort (or hash) and scan a potentially large number of such nodes during their execution. Therefore, the previous method is considered to be ineffective when applied over sparse datasets, especially when these datasets are much larger than the available memory size. The objective of the Partitioned-Cube Algorithm [51] is to extend the GBLP algorithm with a more effective way of partitioning data, in order to reduce I/O and achieve faster computation even when the original fact table is sparse. The Partitioned-Cube algorithm is recursive and follows a divide-and-conquer strategy based on a fundamental idea that has been successfully used for performing complex operations (such as sorting and join). That is, it partitions the large tables into fragments that fit in memory and performs the complex operation on each memory-sized fragment independently. Partitioning into smaller fragments starts from the original fact table and proceeds recursively until all partitions fit in main memory. For each memory-sized partition the Memory-Cube algorithm is called, which computes entire sub-cubes inside memory. At each stage, the Partitioned-Cube algorithm proceeds as follows: It partitions the input table R into n fragments based on the value of some attribute d and recursively calls itself n times passing as input of every recursive call, one partition at a time. The union of the n results is the sub-cube $SC1$ that consists of all nodes that contain d in their grouping attributes. Secondly, it then makes one more recursive call to itself giving as input the more detailed node of $SC1$ and fixing d to the value ALL. In this way, it computes the sub-cube $SC2$ that consists of all nodes that do not contain d in their grouping attributes. The advantage of this algorithm is that it's I/O cost is linear to the number of dimensions. However, when data is sparse, the algorithm might break earlier than expected.

3.1.4 BUC Algorithm

The BUC algorithm [11] was primarily introduced for the computation of Iceberg-Cubes, which computes only those group-by tuples which an aggregate value (e.g count) is above some pre-specified minimum support threshold (minsup). In other words, the algorithm takes into further consideration only sets of tuples that aggregate together, and for which the aggregate function returns a value greater than minsup. The name of the algorithm (BottomUpCube) indicates the way it traverses the cube lattice. It builds the cube in a bottom-up fashion, starting with the computation of the ALL node, which contains only one tuple, moving towards more detailed nodes with more grouping attributes, and finishing at the root of the lattice, which contains all dimensions as its grouping attributes. This feature differentiates BUC from all previous methods, which compute the cube moving top-down, and gives it the advantage of early pruning of tuple sets that do not satisfy the minimum support criteria. Such pruning is similar to the previously introduced Apriori algorithm [3]. That is, the sets of tuples that do not satisfy minimum support at a node cannot satisfy it at any of its ancestors either, and can therefore be pruned. Unlike previously discussed algorithms, to have this pruning ability, BUC sacrifices any exploitation of parent/child commonalities in the cube lattice. This pruning ability makes it particularly effective in the computation of Iceberg-Cube. This is especially true when the original fact table is sparse, since it generates fewer aggregations, making the satisfaction of minimum support more difficult, and thus leading to more pruning.

3.1.5 Condensed Cube Algorithms

A condenced cube [61] is a fully computed cube that condenses those tuples, aggregated from the same set of base relation tuples, into one physical tuple. Let us consider an extreme case as an example. Let relation R have only one single tuple $r(a_1, a_2, ..., a_n, m)$. Then, the data cube for R will have 2^n tuples, $(a_1, a_2, ..., a_n, V_1)$, $(a_1, *, \cdots, *, V_2)$, $(*, a_2, *, \cdots, *, V_m)$, $(*, a_2, *, \cdots, *, V_3)$, $...(*, *, \cdots, *, V_m)$, where $m = 2^n$. Since there is only one tuple in relation R, we have $V_1 = V_2 = ... = V_m = aggr(r)$. Therefore, we only need to physically store one tuple $(a_1, a_2, \cdots, a_n, V)$, where $V = aggr(r)$, in the cube together with some information indicating that it is a representative of a set of tuples. For queries on any cuboid, we can return the value V directly without the need of further aggregation. That is, for this example, the cube for R with 2^n tuples may be condensed into one tuple. In general, tuples from different cuboids in a cube, that are known to be aggregated from the same set of tuples, can be condensed to one tuple. A condensed cube has the following unique features:

- *A condensed cube is not compressed.* Although the size of a condensed cube is smaller than the complete cube, it is not compressed.
- *A condensed cube is a fully computed cube.* It is not based on approaches that reduce the size of a data cube by selectively computing some, but not all, cuboids in the cube. Therefore no further aggregation function is required to answer queries.
- *A condensed cube provides accurate aggregate values.* It does not reduce the cube size through approximation of any form.
- *A condensed cube supports general purpose OLAP applications.* It is different from those proposals to reduce the size of a cube by tailoring it to only answering certain types of queries.

3.1.6 Dwarf Cube Algorithm

A Dwarf Cube [55] is a highly compressed structure for computing, storing, and querying data cubes. A Dwarf cube solves the storage space problem, by identifying prefix and suffix redundancies in the structure of the cube and factoring them out of the store. The main aim of a Dwarf cube [56] is to reduce the storage complexity of data cube to $O(T^{1+1/log_d C})$ where d is the number of dimensions, C is the cardinality of the dimensions and T is the number of tuples. Thus, if we keep the number of tuples in the fact table constant and start increasing the dimensionality of the fact table, the size of the Dwarf increases only polynomially. The first form shows that the dimensionality d is raised to $log_C T$ which is does not depend on d and is actually quite small for most realistic datasets. The second form of complexity shows that the Dwarf size is polynomial with respect to the number of tuples of the data set T. The Dwarf algorithm is the only technique that manages to identify whole sub-cubes as redundant and coalesce the redundancy from both storage and computation time, without calculating any redundant sub-cubes [55]. *Prefix redundancy* can be understood by considering a sample cube with dimensions a, b, c. Each value of dimension a appears in 4 group-bys (a, ab, ac, abc), and possibly many times in each group-bys. For example for the fact table shown in Table 3.1, the value S1 will appear 7 times in the corresponding cube, and more specifically in the group-bys: $(S1, C2, P2)$, $(S1, C3, P1)$, $(S1, C2)$, $(S1, C3)$, $(S1, P2)$, $(S1, P1)$ and $(S1)$. The same also happens with a prefix of size greater than one, where each pair of a, b will appear not only in the ab group-by, but also in the abc group-by. The Dwarf algorithm recognizes this kind of redundancy and stores every unique prefix just once.

Table 3.1: Fact table for Sales cube

Store	Customer	Product	Price
S1	C2	P2	70
S1	C3	P1	40
S2	C1	P1	90
S2	C1	P2	50

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 with a single value a_i of dimension a (such an example exists in Table 3.1, where the value $C1$ appears with only the value $S2$). 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. This happens because the second group-by aggregates all the tuples of the fact table that contain the combination of any value of the a dimension (which here is just the value a_i) with b_j and x. Since x is generally a set of values, this suffix redundancy has a multiplicative effect. Suffix redundancy is identified during the construction of the Dwarf cube and eliminated by coalescing their space. What makes the Dwarf cube technique practical is the automatic discovery of the prefix and suffix redundancies, without requiring knowledge of the value distributions and without having to use sophisticated sampling techniques to figure them out. Dwarf cubes have significant reduction in computation cost by identifying each redundant suffix prior to its computation. Furthermore, because of the condensed size of the Dwarf cube, the time needed to query and update is reduced compared to all previous algorithms. The construction requires a single scan of the fact table. For the first tuple of the fact table, the corresponding nodes and cells are created on all levels of the Dwarf Structure. As the scan continues, tuples with common prefix with the last tuple will be read. The necessary cells required to accommodate new key values as the algorithm progress through the fact table are created. At each step of the algorithm, the common prefix P of the current and the previous tuple is created. The size of the the Dwarf cube is defined as the total number of tuples it contains after data coalescing. The redundancy of the cube is eliminated in the Dwarf cube and coalesced areas are only stored once.

3.1.7 Shell Fragment Approach

The Shell Fragment Approach [39] was specifically designed to specifically compute data cube for high dimensional databases. Since a data cube grows exponentially with the number of dimensions, it is too costly in both computation time and storage space to materialize a full high-dimensional data cube. The Shell Fragment algorithm is based on an observation that, although a data cube may contain many dimensions, most OLAP operations are performed only on a small number of dimensions at a time.

Table 3.2: The original dimension table with five (5) attributes

Tid	A	B	C	D	E
1	a1	b1	c1	d1	e1
2	a1	b2	c1	d2	e1
3	a1	b2	c1	d1	e2
4	a2	b1	c1	d1	e2
5	a2	b1	c1	d1	e3

The Shell Fragment technique, therefore, partitions a high dimensional dataset into a set of disjoint low dimensional datasets called *fragments*. The base dataset is projected onto each fragment, and data cubes are fully materialized for each fragment. With the precomputed *shell fragment cubes*, one can dynamically assemble and compute cuboid cells of the original dataset online. This is made efficient by set intersection operations on the inverted indexes which was previously mentioned. To illustrate the algorithm, consider the sample database in Table 3.2. Let the cube measure be count(). The inverted index is constructed as as follows. For each attribute value in each dimension, we register a list of tuple IDs (*tids*) associated with it.

The inverted index in Table 3.3 may be generalized to multiple dimensions, where one can store tid-lists for combinations of attribute values across different dimensions. For example, suppose we are to compute the shell fragments of size 3. We first divide the 5 dimensions into 2 fragments, namely (A, B, C) and (D, E). For each fragment, we compute the complete data cube by intersecting the tid-lists in Table 2 in a bottom-

Table 3.3: Inverted Indexes for Individual Dimensions A, B, C, D, and E

Attribute Value	Tid List	List Size
a1	1 2 3	3
a2	4 5	2
b1	1 4 5	3
b2	2 3	2
c1	1 2 3 4 5	5
d1	1 3 4 5	4
d2	2	1
e1	1 2	2
e2	3 4	2
e3	5	1

up depth-first order in the cuboid lattice. Thus, to compute the cells $\{a1b2*\}$, the intersection of the tuple ID lists of $a1$ and $b2$ is computed to get a new list {2, 3}. The cuboid for AB is shown in Table 3.4.

Table 3.4: Cuboid for AB

Cells	Intersection	Tuple ID List	List Size
a1 b1	1 2 3 $\cap$1 4 5	1	1
a1 b2	1 2 3 $\cap$ 2 3	2 3	2
a2 b1	4 5 $\cap$1 4 5	4 5	2
a2 b2	4 5 $\cap$2 3	ϕ	0

After computing the cuboid for AB, we may then compute cuboid ABC by intersecting all pairwise combinations between Table 3.4 and the row $c1$ in Table 3.3. Notice that, because the entry $a2b2$ is empty, it can be effectively discarded in subsequent computations based on the Apriori Property [3]. The computed shell fragment indexes are used to facilitate online query computation. The question is how much space is needed to store them. Given a database of T tuples and D dimensions, the amount of memory needed to store the shell fragments of size F is $O(T(\frac{D}{F})(2^F - 1))$. The Shell Fragment algorithm is depicted in Algorithm 1.

The advantage of this algorithm is that, it can drastically reduce the size of the data cube. However, the challenge with this approach is how to select the right fragment size. A fragment size of three (3) has been proposed to provide a good trade-off between storage space and query performance but in real life application, this will result in too many

Algorithm 1 The Shell Fragment Algorithm

Input: A base cuboid B of n dimensions: $(A_1, \cdots, A_n)$.

Output: (1) A set of fragment partitions $P_1, \cdots, P_k$ and their corresponding (local) fragment cubes, and (2) and *ID_measure* array if the measure is not *tuple-count*

```
1: partition the set of dimensions $(A_1, \cdots, A_n)$ into a set of $k$ fragments $P_1, \cdots, P_k$
2: scan base cuboid B once and do the following
3: insert each $(tid, measure)$ into $ID\_measure$ array
4: for each attribute value $a_i$ of each dimension $A_i$
5: build and inverted index entry: $(a_i, tidlist)$

6: for each fragment partition $P_i$
8: build a local fragment cube $S_i$ by intersecting their corresponding tid-lists and computing their
   measures
```

queries computed from more than one fragment and therefore result in poor performance.

3.1.8 Discussion

Data cube computation involves scanning the original data, applying the required aggregate function on all groupings, and generating the cube contents [42]. The 2^D algorithm, represents the first attempt to introduce the idea of data cubes and its computation. It is not an efficient algorithm, with exponential complexity. The GBLP algorithm improves the performance of the 2^D algorithm by computing each node in the lattice using its smallest parent, not the original fact table. The main disadvantage of this method is that it only prunes the lattice into a tree, without suggesting a particular way to traverse it. Lack of an efficient disk-access plan makes GBLP impractical when dealing with large datasets that do not fit in memory. The Partition-Cube algorithm is based on the assumption that real world data tends to be sparse, and this sparsity results in fewer aggregation operations. The Partitioned-Cube algorithm is recursive and follows a divide-and-conquer strategy based on a fundamental idea that has been successfully used for performing complex operations such as sort and join. Unlike the 2^D and $GBLP$ algorithms which are exponential in nature, the I/O cost of Partitioned-cube is typically linear to the number of dimensions. However, when sparsity increases as a result of data increase, the number of redundant tuples also increases, leading to an earlier recursion break. The BUC technique, however, takes into consideration only sets of tuples that aggregate together, and for which the aggregate function returns a value greater than minsup. The distinguishing feature of the BUC method is that it starts computation from the apex (ALL) node, which contains only one tuple, moves towards more detailed nodes with more grouping attributes, and finishes at the root of the lattice which contain all dimensions as its grouping attribute. The BUC technique performs very well even in sparse environment. However, it has shown to suffer from the curse of dimensionality. The Condensed Cube method aims to reduce the size of a data cube and hence its computation time and storage overhead. It yields a fully computed cube that condenses those tuples aggregated from the same set of base relation tuples into one physical tuple. It has the advantage of reducing the size of the cube. However, it suffers from the curse of dimensionality, since it creates a full cube. Another approach that aims to reduce the storage complexity of a data cube is the Dwarf cube algorithm. A Dwarf cube eliminates prefix and suffix redundancy by factoring them out of the data store. The Dwarf cube algorithm has the advantage of a polynomial size and computation complexity with respect to dimensionality increase. The disadvantage is that, when the cardinality increases with a corresponding increase in dimensionality the storage and computation complexity becomes exponential.

Table 3.5: Comparison of data cube computation approaches

Algorithm	**Advantages**	**Disadvantages**
2^DAlgorithm	It is very simple algorithm	
GBLP	More efficient than 2^DAlgorithm	Only prunes the lattice into a tree, without suggesting a particular way to traverse it
Partition-Cube	I/O cost is linear to number of dimension	Sparsity in data may lead to earlier recursion break
BUC	Performs well, even when data is sparse	Suffers from curse of dimensionality
Condensed Cube	Reduces the size of data cube drastically	Suffers from curse of dimensionality
Dwarf Cube	Polynomial size and computation complexity with respect to dimensionality increase	Exponential complexity with respect to both dimensionality and cardinality increase
Shell Fragment	Linear complexity with respect to dimensionality increase	Does not perform well for larger fragment sizes

Finally, the Shell Fragment method attempts to reduce the size of the data cube as dimensionality grows by partitioning the dimension space into small disjoint sets called fragment. A full data cube is then computed for each fragment. The advantage of this method is that, for a fixed fragment, increase in dimensionality only means additional fragments. Thus the relationship between storage size and dimensionality increases linearly. However, this method only works for small fragment sizes of 2 and 3, which is too small for most real world queries. Furthermore, increasing the fragment size tends to cause an exponential explosion in size. Table 3.5 summarizes the discussion with a comparative study of all the techniques.

Apart from data cube computation, data selection methods (also referred to as the view selection problem) is essential for an efficient data cube, since it determines which part of the data cube will be materialized. Next, we discuss data selection methods.

3.2 Selection Methods

Recall that data cube Selection Methods (also known as the View Selection Problem (VSP)) determine the subset of the data cube that will actually be stored [42]. Thus, given a certain storage cost threshold, there is a need for selecting the best views to be materialized, i.e views that fit the storage requirement and provide the lowest response time to process OLAP queries. A data cube selection method takes as input a fact table and returns a set of views to store in order to speed up queries [29]. These methods are also referred to as cube materialization techniques. Recall that the performance of cube selection techniques are usually dependent on three criteria, namely (1) the amount of memory to store the selected view, (2) the query response time and finally (3) the time complexity of the algorithm. As mentioned earlier, most of the works proposed in the literature consider the problem of finding the *best* data to store in order to optimize query evaluation time while the memory space needed by this data does not exceed a certain limit fixed by the user. Their objective is thus usually to balance the trade-off between the amount of resources consumed by the data cube and query response time. The problem of selecting an optimum subset of a data cube is NP-complete, even under some rather loose constraints [30]. Hence, all methods search for near optimum solutions using appropriate heuristics. In effect, an effective selection method in general accelerates computation as well, since partial cube materialization implies smaller cost of writing the result to the disk and probably fewer aggregation operations. All the cube selection methods can be categorized into five (5) main techniques. These are the techniques based on greedy heuristics, random search heuristics, evolutionary heuristics, hybrid heuristics and model-driven heuristics. In this section we will study the Greedy heuristics proposed in the literature which is the focus of our work.

3.2.1 Greedy Algorithm

The Greedy Algorithm [30] represents the first systematic attempt to find a solution to the problem of selecting an appropriate subset of a cube for computation and storage. It is based on a "greedy" heuristic in order to avoid an exhaustive search in the space of all possible solutions. The Greedy algorithm provides a solution to the selection problem with the goal of minimizing the average time taken to evaluate a lattice node on-the-fly, under the constraint that the number of nodes that are maintained in materialized form is bounded. The space used by the materialized nodes plays no part in this approach. This optimization problem is NP-complete [30] even in this simple setting. The algorithm assumes that the storage $cost(size)$ of each node is known a-priori. Let $C(u)$ be the size of node u, measured in terms of the number of tuples. Suppose also that there is a limit k in the number of nodes that can be stored along with the root of the lattice. The root is always materialized, since it contains the most detailed information and cannot be computed from it's dependents. Let S be the set of nodes that the algorithm selects. Set S is initialized to the root of the lattice. The algorithm is iterative and terminates after k iterations. In each iteration, one more node is added to S, building the final result in steps. The choice of a node u in the i^{th} iteration depends on the quantity $B(u, S)$, which is called the benefit of u relative to S and is calculated as follows.

1. For each node $w \subseteq u$, where $w \subseteq u$ denotes that w is either u or one of its descendants, let v be the node of least cost in S such that $w \subseteq v$. (Note that such a node always exists and in the worst case it coincides with the root.) Then, define the quantity B_w as $B_w = max\{C(v) - C(u), 0\}$.

2. $B(u, S)$ is defined as $B(u, S) = \sum_{w \subseteq u} B_w$.

In other words, the benefit of u relative to S depends on the improvement that its materialization (addition to S) offers in the cost of computing itself and all of its descendants. For each $w \subseteq u$, the cost of computing w using u is compared with the cheapest cost of computing w using some node v that already belongs to S. If u aids, which means that $C(u) < C(v)$, then difference $C(v) - C(u)$ contributes to the total benefit, which is the sum of all such differences. In the i^{th} iteration, the Greedy algorithm inserts to S the node u that does not belong to it and has the maximum benefit $B(u, S)$.

The complexity of the Greedy algorithm is shown to be $O(k \times n^2)$, where n is the number of lattice nodes. Furthermore, the authors of Greedy proved that, regardless of the given lattice, the total benefit of the Greedy is at least 0.63 times the total benefit of the optimal solution. This, however, does not necessarily imply near-optimal performance. Nevertheless, it is an indication of certain guarantees in the effectiveness of the algorithm.

3.2.2 BPUS Algorithm

The BPUS algorithm [30] is a variation of the Greedy algorithm. The difference between the BPUS and Greedy algorithms, lies in the constraint that must be satisfied. Instead of having a limit on the number of nodes that can be materialized, there is an upper bound on the total storage space that the precomputed nodes can occupy. Again, this upper bound does not include the space occupied by the lattice root, which is always stored. The fact that BPUS takes into account the storage requirement of an OLAP server gives a more realistic criterion, since in practice, storage is usually a limiting factor. BPUS functions in the same way as the Greedy algorithm, proceeding iteratively and adding nodes to the set S, while the upper bound of disk space occupied by the nodes in S has not been exceeded. The only difference is in the way a node u is selected in the i^{th} iteration. The choice is not based on the absolute benefit $B(u, S)$, but on the benefit per space-unit that u occupies - BPUS selects the node that does not belong to S and maximizes the fraction $B(u, S)/C(u)$. A problem arises when there is a very small node s with great benefit per space-unit and a much larger node b with similar benefit per space-unit. Selecting s may exclude b from further selection, since it might no longer fit. Nevertheless, BPUS also provides performance guarantees, similar to the Greedy approach. That is if no lattice node occupies space larger than some fraction f of the totally allowed storage space, then the total benefit of the solution of BPUS will be at least $(0.63 - f)$ times the total benefit of the optimal solution.

3.2.3 Greedy-Interchange and Inner-Level Greedy algorithm

The general problem of choosing a set of views for materialization in a data warehouse environment, given some constraint on the total space that can be used, in order to optimize query response time was studied in [22]. The proposed algorithms are based on a priori knowledge of queries that will be executed, their frequencies, and the cost of updating the materialized views. However, the problem the authors state is more general than the problem solved by the Greedy algorithm and the BPUS algorithm, since they focus on the selection of nodes of the cube lattice, not only on the selection of any view that can be materialized. The authors proposed two new algorithms, namely the Greedy-Interchange algorithm and the Inner-Level Greedy algorithm.

The Greedy-Interchange algorithm starts from the solution generated by the BPUS algorithm and improves it by exchanging nodes that have not been selected with selected nodes. Since the nodes have difference sizes, it is possible to exchange one or more nodes with one or more other nodes. The algorithm iterates until there is no possible exchange that improves the total benefit. Unfortunately, nothing has been proven about the effectiveness of this algorithm, except for the obvious fact that it constructs a solution at least is as good as the BPUS algorithm. Moreover, a great disadvantage of the Greedy-Interchange algorithm is that there are no bounds on its execution time. In the paper, the authors mention that in a great number of experiments, the algorithm usually terminates after some time that is at most a factor 1.5 greater than the execution time of BPUS. However, no strict bound has been theoretically proven. The Inner-Level Greedy algorithm, however, takes into account the existence of indexes on the selected nodes, which can also be stored if they positively affects the total benefit. Thus, the algorithm selects not only nodes but indexes as well. The Inner-Level Greedy algorithm is also iterative. In each iteration, it selects a subset C that may consist of either one node and some of its indexes in a greedy manner or a single index of a node that has already been selected in some previous iteration. In particular, the iteration consists of two steps:

1. In the first step, for each node v_i, a set IG_i is constructed and is initialized to v_i. Then, the indexess of v_i are added one by one to IG_i in order of their incremental benefits, until the benefit per unit space of IG_i with respect to S is maximized. Here, S denotes the set of nodes and indexes already selected. The algorithm chooses as C the IG_i that has the maximum benefit per unit space with respect to S.

2. In the second step, the algorithm selects an index of a node that already belongs to S. The selected index is the one whose benefit per unit space is the maximum with respect to S.

The benefit per unit space of C from first step is compared with that of the index selected in the second step. The better one is added to S.

The complexity of the algorithm is $O(k^2 \times n^2)$, where k denotes the maximum number of nodes and indexes that fit in the space given according to the initial constraint, and n is the total number of nodes and indexes among which the algorithm selects. It has been proven that the total benefit of Inner-Level Greedy is at least 0.467 times the total benefit of the optimal solution.

3.2.4 MDred-lattice Algorithm

The MDred-lattice algorithm selects an appropriate subset of lattice nodes so that the system responds efficiently to certain known queries, balancing at the same time the space that this subset occupies on disk [8]. This implies that the algorithm optimizes the average response time of queries in a particular workload and is not concerned with the overall average query. Given a set of queries SQ as the workload of interest, the algorithm defines the set of the so-called candidate views, which contains the lattice nodes that will be beneficial if saved. A lattice node V_i belongs to the candidate-view set, if it satisfies one of the following two conditions:

1. Node V_i is associated to some query $q_i \in SQ$, in particular, the one that it contains in its group-by clause, the same grouping attributes as V_i

2. There are two candidate views V_j and V_k, which have V_i as the most specialized common ancestor.

Since the number of candidate views is too large in practice, the authors proposed a variation of the algorithm that takes into account the estimated size of each candidate view as well [42]. Depending on the sizes of the candidate views, it is decided whether or not a level of the lattice is too specialized so materialization of its nodes does not offer great benefit for answering the queries of interest. In this case, the nodes of the selected level are substituted by some of their ancestors in higher levels.

3.2.5 PBS Algorithm

The PBS algorithm [53] is very similar to the BPUS algorithm, but uses a simpler heuristic for solving the problem of selecting a cube subset for storage. As in the BPUS algorithm, an upper bound on the total storage space that precomputed nodes can occupy must be satisfied. PBS initializes some set S to the root of the lattice and then iterates, each time selecting and adding into S the smallest remaining node, until it exceeds the space limit posed by the initial constraint. Interestingly, the PBS algorithm proceeds in a bottom-up fashion, traversing the lattice from smaller and more specialized nodes towards larger and more detailed ones. Proceeding this way, the algorithm creates a frontier that separates the selected nodes from the unselected ones. Recall that the lower a node is in the lattice, the smaller it it, since the number of aggregations it has been subject to increases from level to level. The algorithm is rather simple, since it calculates no cost and does not take into account a benefit quantity like the Greedy and BPUS algorithms. It only needs an estimation of the size of each node. Hence, its complexity is $O(n \times \log n)$, where n is the number of lattice nodes. The factor $n \times \log n$ comes from ordering the nodes by their sizes. Figure 3.2 shows the lattice of a 4 dimensional cube and the estimated size of each node in blocks. Let the available storage space be 200 blocks. The nodes that are selected for storage have been shaded. The dashed line denotes the frontier that separates the selected from the nodes not selected. Note that 16 blocks remain unexploited. The main disadvantage of the algorithm is that it sacrifices the quality of its results in order to accelerate the process. In particular as for BPUS algorithm, the total benefit of the PBS solution is at least within a factor of $(0.63 - f)$ of the optimal algorithm total benefit (where f is the maximum fraction of the size occupied by a node with respect to the total available space). For the PBS algorithm, however, this property holds only under strict conditions. More precisely, such performance guarantees can be given only for the subclass of Size Restricted hypercube lattices (or SR-hypercube lattices), which have the following special ordering between the sizes of parent and children nodes. For each pair of nodes u, w, where w is the parent of u and different from the root, if k denotes the number of children of w, then $|u|/|w| \leq 1/(1 + k)$. In other words, in SR-hypercube lattices, the size of a node is equally distributed to its children, which have to be at least $(1 + k)$ times smaller than their parents.

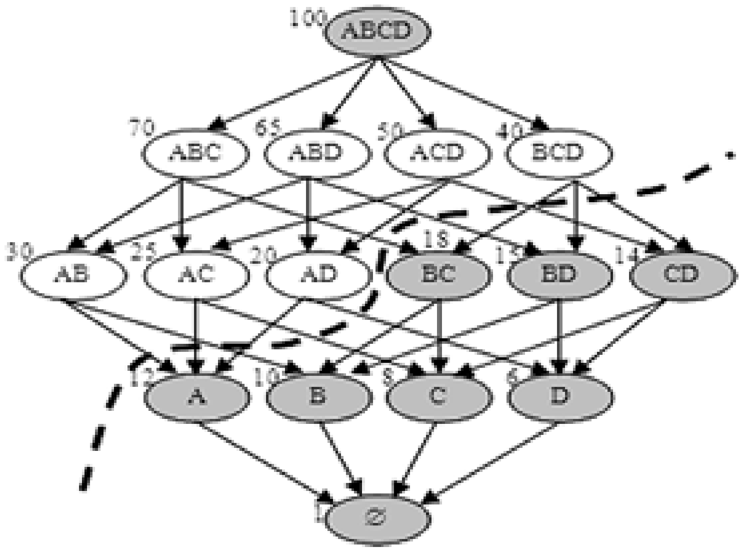


Figure 3.2: The nodes selected by PBS [42]

3.2.6 Inverted-Tree Greedy Algorithm

Most of the algorithms presented up to this point provide solutions to the view selection problem under a space constraint. Nevertheless, since the price/capacity ratio of secondary storage media keeps decreasing, the limiting factor during the view selection process is not always the space that is available for the storage of the selected materialized views; it can also be the time necessary for their incremental maintenance, a process typical in data warehouses.

The view selection problem under a maintenance-cost constraint seems similar to the view selection problem under a space constraint, except for one fundamental difference [24] that makes it harder to solve. The space occupied by a set of views monotonically increases when more views are added to the set, whereas the maintenance cost does not. In other words, it is possible that the maintenance cost of a set of views decreases after the insertion of one more view in the set. For example, if the latter is an ancestor of some views in the original set and its updated data can be used as intermediate results to accelerate the update process of its descendants. However, because of the non-monotonic property of the update cost, a straightforward greedy algorithm, in the spirit of BPUS algorithm, may produce a result of arbitrarily poor quality compared to the optimal solution [24]. Inverted-Tree Greedy [24] is an algorithm that overcomes this problem.

The Inverted-Tree Greedy algorithm uses a measure called *benefit per unit of effective maintenance cost* (referred to as benefit hereafter), which is similar to the benefit per space-unit used by BPUS algorithm, but which takes into account maintenance costs instead of storage costs [24]. As mentioned above, incrementally building the final solution by greedily choosing the lattice node with the largest benefit with respect to the set S of nodes already selected for materialization has been found to produce results of poor quality. Alternatively, the Inverted-Tree Greedy algorithm iteratively selects the most beneficial *Inverted-Tree set* with respect to S, consisting of a set of nodes P of the cube lattice that do not belong to S. A set of nodes P is defined to be an inverted-tree set in a directed graph G (in our case, the cube lattice) if there is a subgraph T_p in the transitive closure of G such that the set of vertices of T_p is P and the inverse graph of T_p is a tree.

In other words, the Inverted-Tree Greedy algorithm works iteratively while the maintenance cost constraint is satisfied. At each step, it considers all possible inverted-tree sets of the cube-lattice nodes that do not belong to S. Among them, it selects the one with the largest benefit with respect to S and inserts its nodes into S. Note that, unlike the greedy methods presented above, in each step, the Inverted-Tree Greedy algorithm selects a set of nodes instead of a single one.

It has been proven [24] that under certain conditions such as view graph instances, which usually hold in practice, Inverted-Tree Greedy returns a solution S whose total benefit is a least 0.63 times the total benefit of the optimal solution that incurs at most the same maintenance cost. Unfortunately, in the worst case, the total number of inverted-tree sets in a directed graph is exponential in the size of the graph, which increases the computational costs of the method, raising some concerns about its applicability in real-world applications.

3.2.7 PGA Algorithm

The PGA algorithm [44] uses a heuristic that is polynomial to the number of dimensions D. The PGA algorithm is iterative, with every iteration divided into two phases, nomination and selection. During nomination, PGA selects a path of D promising nodes from the root to the bottom of the lattice. The nodes in this path are the candidates considered for materialization in the second phase. The nomination process then moves down one level. The nomination phase begins with the root as the parent node and estimates the sizes of all its children nodes. The smallest child that has still not been selected during previous iterations is nominated as a candidate view. The process continues until the bottom of the lattice is reached. When the nomination phase completes, the PGA technique moves to the selection phase, during which it selects for materialization the candidate view with the largest estimated benefit. The benefit of materializing a candidate view is estimated based on some metadata and not calculated by recomputing the query response time of all its descendants, as performed by BPUS algorithm. The PGA algorithm avoids the two main sources of exponential complexity with respect to D of all previous algorithms. First, during all iterations, it does not consider all remaining nodes on the entire lattice, which are in the order of 2^D, but only D candidate nodes. Second, it does not calculate the benefit of a candidate node by visiting all its descendants, which are again in the order of 2^D in the worst case, but it estimates this benefit in $O(1)$ time based on specialized metadata. The authors show that the time complexity is $O(K \times D^2)$, where K is the number of nodes finally selected.

3.2.8 PickBorders Algorithm

The PickBorders algorithm [29] propose a solution that gives a good trade-off between memory usage and queries cost with reasonable time complexity. The aim of the algorithm is to reduce the size of the data cube with a query performance guarantee. The authors address the following problem. Given a real number $f \geq 1$, their aim is to find a set of cuboids S of minimum size so that $cost(S) \leq f \times mincost$. Thus, they aim to name a set S of cuboids which, when materialized, the evaluation cost of queries does not exceed f times the minimal cost. Moreover S should be minimal size. Clearly, if $S = C$ then this solves the problem. However, this solution is unrealistic because the size of C is huge. Thus, a constraint is added to the problem to facilitate the selection of the smallest S such that $Cost(S) \leq MinCost \times f$. The authors presented three systematic algorithms for selecting a subset of views to be stored. The first algorithm *Pick Small Cuboid* (PSC), considers the situations where the size of the cuboids are known or unknown a priori. In the case where it is known, computing the subset of views S involves sorting the views and selecting those whose size is less than M/f. On the other hand, if the size is unknown, then the sizes of all cubes can be computed before sorting and selecting views that are less than M/f. Note that M is the Size of the base cuboid. However, the condition $size(c) \leq M/f$ is anti-monotonic, that is if $c \leq c'$ and $size(c) > M/f$ then $size(c') > M/f$. That implies that the algorithm for selecting S is exactly the Apriori algorithm [3]. Below is the detailed algorithm for PSC.

Algorithm 2 Pick Small Cuboid Algorithm

Input: Parameter f, fact table T.

Output: a partial data cube S

```
1: $S = \phi$
2: $C_1$={$c \in C_1$ s.t $|c| \leq M/f$}
3: $S = S \cup L_1$
4: for($i = 1; L_i \neq \phi; i++$) do
5: $C_{i+1}$={candidates generated from $L_i$}
6: for each $c \in C_{i+1}$ do $compute\_size(c)$
7: $L_{i+1}$={$c \in C_{i+1}$ s.t $|c| \leq M/f$}
8: $S = S \cup L_{i+1}$
9: Return $S$
```

A problem with the PSC algorithm is that the size of S might be too big and thus could not be stored entirely. In order to reduce the size of S, the algorithm identifies and stores only that subset of cuboids that can reduce the maximal cost of computing them by a factor f. Thus, if only the *maximal cubooids* w.r.t f are stored, then the maximal cost of cuboids are reduced. To reduce the maximal cost of cuboids, the authors identified and eliminated the set of cuboids for which reduction is possible with respect to f. This set is called the *f-Reducible Set of Cuboids*. A cuboid c is said to be f-reducible if it can be computed from a cuboid whose size is less than the maximal cuboid divided by the factor f.

For example, in Figure 3.3, the cuboid ABDE is not 10-reducible because none of

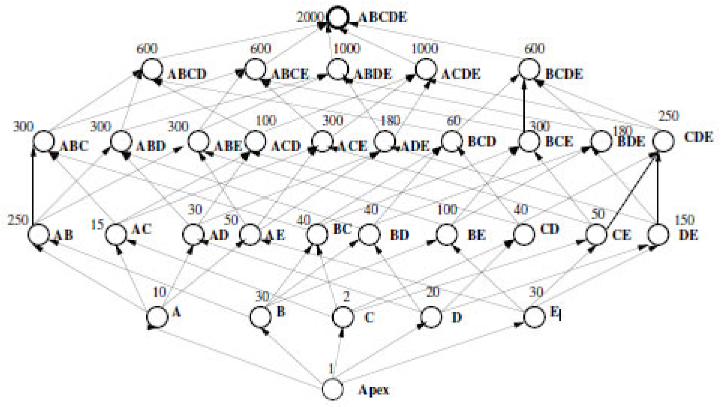


Figure 3.3: Sample data cube [29]

its ancestors has a size less than $Size(ABCDE)/10$. The cuboid B is 10-reducible because at least one of its ancestors has a size less than $size(ABCDE)/10$. Although this improvement to the PSC algorithm reduces the space consumed by PSC, it does not guarantee query performance. In order to guarantee the quality factor and reasonable memory size, the solution is improved by iterating through the algorithm for different values of f, that is f^i where i is an integer ranging from 0 to $\lfloor log_f(M) \rfloor$. The final algorithm is shown below.

```
1: $S = \phi$
2: for($i = 1; i \leq \lfloor log_f(M) \rfloor ; i++$) do
3: $S = S \cup PSC(f)$
4: $f = f^i$
5: Return $S$
```

PickBorders guarantees two properties. First, the cost of its output S is bounded by the minimal cost times the factor f. This property may be qualified as a global property. Secondly, the same output guarantees that if we take each cuboid individually, its cost with respect to S is also bounded by the minimal cost times f.

3.2.9 DynaMat Algorithm

All selection algorithms previously described are static in the sense that, given some restriction regarding the available space and/or update time, and sometimes some description of the expected workload, they statically suggest a set of views to materialize for better performance. Nevertheless, decision support data are dynamic, since both data and trends in it keep changing. Hence, a set of views statically selected might quickly become outdated. Keeping the set of materialized views up to date requires that some of the aforementioned view-selection algorithms may be rerun periodically, which is a rather tedious and time-consuming task. The DynaMat algorithm overcomes this drawback by constantly monitoring incoming queries and materializing a promising set of views subject to space and update time constraints [36]. Another strong point of the DynaMat algorithm is that it does not materialize entire views, but only the segments of them that correspond to the results of aggregate queries. This is achieved using certain types of range selection queries on the dimensions of interest. Hence, this algorithm performs selection at a finer level of granularity. The segments are stored in a so-called *view pool*, which is initially empty. The *view pool V* is the information repository that is used for storing materialized results. The algorithm distinguishes two operational phases, namely on-line and update. During the online phase the system answers queries. When a new query arrives, the DynaMat algorithm uses a *Fragment Locator* to determine if it can answer it efficiently using a segment already stored in the pool. A *Directory Index* implemented using R-trees is maintained in order to support sub-linear search in the view pool for finding candidate materialized results. R-trees are balanced search trees that group nearby objects and represent them with a minimum bounding rectangle in the next higher level of the tree. Interested readers are referred to [41] for a detailed discussion. Otherwise, it accesses the source data stored in the fact table (and potentially some relevant indexes) to answer the query. This decision is based upon a cost model that compares the cost of answering a query through the repository with the cost of running the same query against the warehouse. In both cases, after computing the result, the DynaMat algorithm invokes an admission control module to decide whether or not it is beneficial to store the result in the pool. If the result fits in the pool, the admission control module always permits its storage for future use. Otherwise, if the pool size has reached its space limit, the admission control module uses a measure of quality (e.g., the time when a segment was last accessed) to locate candidate segments for replacement. A segment already stored in the pool is considered for replacement only if its quality is lower than the quality of the new result. If no appropriate replacement candidates are found, the admission control module rejects the request for storage of the new result in the pool. Otherwise, the approach applies some replacement policy (e.g., Least Frequently Used (LRU)) to replace some old segments with the new one. Thus, the algorithm guarantees that the size of the pool never exceeds its limits. During the on-line phase, the goal of the system is to answer as many queries as possible from the pool, because most of them will be answered quicker from the view pool than using conventional methods, for example, using the source data warehouse. At the same time, it quickly adapts to new query patterns and efficiently utilizes the system resources.

During the update phase, the system receives updates from the data sources and refreshes the segments materialized in the pool. In order to guarantee that the update process will not exceed the available update-time window, the DynaMat algorithm estimates the time necessary for updating the materialized segments. If this exceeds the system constraints, it again uses a policy based on the chosen measure of quality to discard some segments from the pool and decrease the total time necessary for the update process. Initially, it discards all segments whose maintenance cost is larger than the update window. Subsequently, it uses a greedy algorithm that iteratively discards segments of low quality until the constraint is satisfied. (As already mentioned, discarding some materialized data does not necessarily imply a decrease in the overall update time. The algorithm takes this into account by not considering segments that contribute to the acceleration of the update time of other segments for replacement.) The remaining segments are updated and the system returns to the on-line phase.

3.2.10 Discussion

Selection Methods selects the *best* views to be materialized that would fit storage requirement and provide the lowest response time. The Greedy algorithm represents the first systematic attempt. Its goal is to minimize the average time taken to evaluate a lattice node "on-the-fly", under the constraint that the number of nodes that are maintained in materialized form is bounded. A disadvantage of this method is that it assumes the storage size of each node is known apriori. Also, the size can explode as it is not bound by storage size. A variation of the Greedy algorithm called the BPUS algorithm, puts an upper bound on the total storage space, instead of having a limit on the number of nodes that can be materialized. The drawback of this algorithm is that, when there are very small nodes with great benefit per space-unit and much larger nodes with similar benefit per space-unit, selecting the smaller views may exclude the larger views, since there may be no space. The Greedy-Interchange and Inner-Level Greedy algorithms are based on apriori knowledge of queries that will be executed, their frequency, and cost of updating the materialized views. Unlike the other approaches discussed so far, the MDred-lattice algorithm is not concerned with overall average query performance, but optimizes the average response time of queries in a particular workload. The objective of Mdred-lattice algorithm is to select an appropriate subset of lattice nodes so that the system responds efficiently to certain known queries, under the constraint of storage space. The problem with this technique is the huge number of candidate nodes that are generated. The PBS algorithm is similar to the BPUS algorithm, but with a simpler heuristic. The PBS algorithm initializes some set S to the root of the lattice and then iterates, each time selecting and adding the smallest node into S until space limit is exceeded. Although this algorithm is very fast and simple, it sacrifices the quality of the results.

Table 3.6: Comparison of data cube selection approaches

Algorithm	**Advantages**	**Disadvantages**
Greedy	Guarantees a certain level of effectiveness by avoiding exhaustive search in all spaces of all solutions	Size of selected cube might exceed space available
BPUS	Storage space is never exceeded	Some beneficial nodes might be eliminated as a result of space constraint
Greedy-Interchange & Inner Level Greedy-Interchange	Focuses on selecting nodes of a cube lattice and not any view that can materialized	There is no bound on the execution time
MDred-lattice	Performs very well on workload queries	Not concerned with overall average query
PGA	Use of specialized meta data the reduce the time of visiting the candidate nodes from O(2D) to O(1)	Might be time consuming and might select a very large number of views
PickBorders	Guarantees a constant approximation factor of query response time with respect to the optimal solution	Difficult to select the right value for f the performance factor
DynaMat	A dynamic solution that relieves the warehouse administrator from having to monitor and calibrate the system constantly	Complicated and time consuming, especially for large databases

Unlike other algorithms, the Inverted-Tree greedy algorithm is based on the premise that price of storage keeps decreasing, therefore storage constraint is not an appropriate limiting factor. The Inverted-Tree Greedy algorithm uses the time necessary for incremental maintenance as the limiting factor. The algorithm works iteratively, by selecting views that ensures that the maintenance cost constraint is satisfied. The disadvantage of this approach is that, in the worst case, the number of Inverted-Tree set in a directed graph is exponential to the size of the graph, thus increasing the computational cost of the method. A peculiar problem in all the algorithms discussed so far is that they have exponential complexity with respect to the number of dimensions D. The PGA algorithm, therefore, avoids two sources of exponential complexity with respect to D of all the previous algorithms. Firstly, during its iterations, it does not consider all remaining nodes on the entire lattice but only a path of D candidate nodes. Secondly, it does not calculate the benefit of a candidate node by visiting all its descendants. The disadvantage is that, since it keeps selecting the smallest child at every iteration, it has the same performance challenge as the PBS algorithm. All the methods discussed so far does not guarantee query performance. The PickBorders algorithm selects views that guarantees a constant approximation factor of query response time with respect to optimal solution to be stored. The algorithm guarantees that the cost of its output S as well as the cost of each individual cuboid is bounded by the minimal cost times a factor f. This is the only systematic algorithm that provides such guarantees. It computes its solution by sorting the cuboids and selecting those whose size is less then the M/f, where M is the size of base cuboid and f is an input parameter. Unlike all the algorithms discussed so far, which are static, the DynaMat algorithm on the other hand is dynamic. It constantly monitors incoming queries and materializes a promising set of views subject to space and update time constraint. It is based on the premise that, if a query result is not wasted by discarding it, its generation cost could be amortized over multiple uses of the result. This algorithm unifies the view selection and the view maintenance problems under a single framework. During updates, the DynaMat algorithm reconciles the current materialized view selection and refreshes the most beneficial subset of it within a given maintenance window. The main benefit of the DynaMat algorithm is that it represents a complete self-tunable solution that relieves the data warehouse administrator from having to monitor and calibrate the system constantly. However, the task of automatically monitoring constantly the query pattern and periodically recalibrating the materialized views is rather complicated and time consuming, especially in large data warehouses where many users with different profiles submit their query. Table 3.6 shows a comparison all of all the data cube selection approaches.

3.3 Summary

In this chapter, we covered some of the main techniques used in data cube computation and selection. The concept of data cube computation was explained, and some of the state-of-the-art computation algorithms were discussed. After generating the cube content, there is the need to select the subset of data that will actually be stored. This problem is referred to as the data cube selection problem. We studied the problem and presented some of the solutions presented in the literature.

In the next chapter, we will introduce our algorithm for computing Personalized Smart Cubes. .

Part II

Personalized SMART CUBES for scalable, fast data access

Chapter 4

The Personalized Smart Cube Approach

Recall that, in OLAP applications, a main focus is to optimize query response time. To do so, we often resort to precomputing or materializing query results. However, due to space or time limitations, we cannot store the result of all queries. So, one has to select the best set of queries to materialize. In the multidimensional model, more precisely when considering data cubes, the relationships between the views can be used in order to define what the best set of views are. Another challenge is that the source data from which data cube has to be computed are usually very large, with high dimensionality. However, users are not interested in the entire data space. Rather, they are only interested in that small subset that is of most interest to them. This user interest is not static and therefore precomputing the personalized cube in advance for each user is not a practical solution. In order to meet such impromptu changes in user interest, we have to constantly monitor user behavior and personalize the data cube accordingly.

In this chapter we present our Personalized Smart Cube approach, which has been developed to address the above-mentioned challenges. We start by presenting a formal definition of the problem in Section 4.1. The details of the Personalized Smart Cube algorithm is explained in Section 4.2 with a running example. The second part of the algorithm, which focuses on the personalization of the Smart Cube, is presented in Section 4.3. After cube computation and personalization, we examine how queries are executed on the personalized Smart Cube in Section 4.4. Finally we conclude with a summary in Section 4.5.

4.1 Problem Definition

Given a fact table T defined as a relation where the set of attributes is divided into two parts, namely the set of dimensional key attributes D_i and the set of measures M_i. In general, D_i is a key of T. A data cube built from T is obtained by aggregating T and grouping its tuples in all possible ways. That is, assuming that each c corresponds to a cuboid, we perform all *Group By* c where c is a subset of D_i. Let the data cube computed from T be denoted by C. Let the dimensions of the data cube be denoted by Dim. If C is a data cube, and D_i is its dimensions, then $|D_i| = D$. The set of cuboids of C is denoted by v. Clearly $|v| = 2^D$. The fact table T is a distinguished cuboid of v and it is called the base cuboid denoted by c_b. The size of a cuboid c is expressed by the number of its rows and is denoted by $size(c)$. The size of a set of cuboids S is denoted by $size(S)$. Note that, if $s \preceq w$ then cuboid s can be computed from cuboid w.

Assume the queries that are asked against C are all and only those of the form *SELECT * FROM c* or *SELECT * FROM T GROUP BY c* where c is a cuboid from C. There are three situations that needs to be considered here. The first one is where only the base cubiod c_b is stored (materialized). In this case, every query requires the use of this cuboid and hence has a time cost that is proportional to the size of c_b. The second situation is the one where all cuboids are materialized. In this latter case, the evaluation of each query consists of first locating the appropriate cuboid and then scanning the corresponding cuboid, making its cost proportional to the actual size of the query or cuboid. However, in real world situations memory space is limited and therefore we cannot feasibly store all the 2^D cuboids, or we do not have time to compute the entire data cube. Rather, what is often done is a partial materialization, i.e. the third case. The personalized smart cube algorithm we present next is based on smart materialization, which is a partial cube materialization algorithm that select only the views that guarantees a fast query response time.

4.2 Personalized Smart Cube Approach

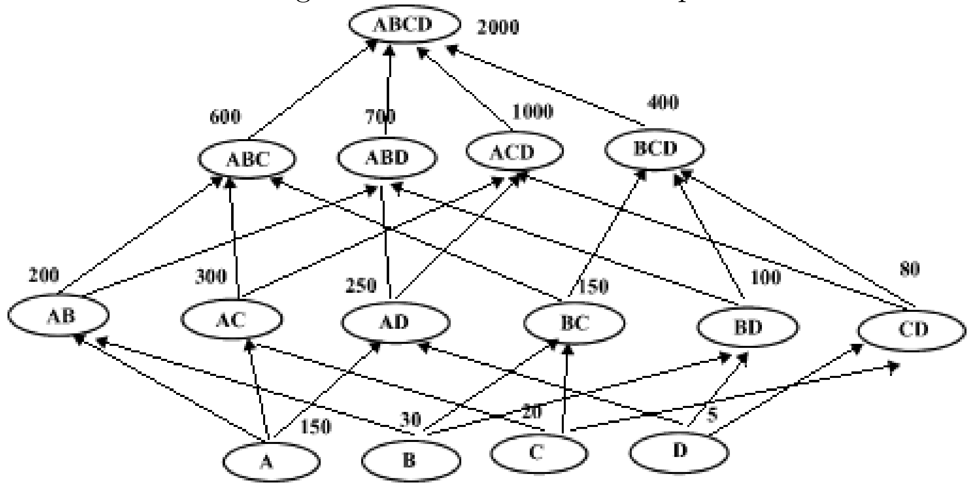
From the above motivation, we propose a novel algorithm called the *Personalized Smart Cube*. The Personalized Smart Cube algorithm uses three main techniques to reduce the size of the data cube while guaranteeing query performance and catering for user interest. These three techniques are data partition, partial materialization and dynamic view materialization. The general idea is that, we first partition the multidimensional data mart into disjoint set of views called fragments. A fragment is a view constructed from dimensional attributes from one or two dimension tables and measure attributes from the corresponding fact table. After the partitioning, each fragment will be a view constructed from one or two dimension tables with the measure values that correspond to the dimension tables. Next we compute a local cube from each fragment, using partial data cube materialization, where each fragment view is considered as a base table.

The view selection algorithm for the partial cube materialization extends the Pick-Borders algorithm as presented in Section 3.2. In our work, we select only cuboids whose sizes are at most M/f and eliminate cuboids whose sizes are below the row threshold. Note that, M is the size of the base cuboid of the fragment, and f is an input factor called performance threshold. The row threshold is the minimum size of a cuboid (or view) that gives an acceptable response time, such that any improvement in query response time as a result of cuboids with tuples below this threshold is considered insignificant. The row threshold is based on the premise that the size of a database table determines its query response time [31]. Thus, smaller views with few tuples takes shorter time to query than larger views with a large number of tuples.

Let us consider the following example as depicted in Figure 4.1. Assume the row threshold is 200. This implies that the time required to process a table of 200 tuples is the minimum acceptable time. Assume also that $f = 4$, that is we select only views whose sizes are below 2000/4. Thus, we first eliminate all cuboids with size below 200 to reduces the search space for phase two of the algorithm where we select only cuboids whose sizes are below 2000/4. The cuboids that meet these two conditions are retained using inverted indexes. After the partitioning and view selection for local cube computation, the size of the overall data cube computed is small, but with some drawbacks. The major drawback is that user queries are mostly unpredictable and in most cases these queries might be required to be computed from attributes of different partitions. In such a situation, an online computation is required to answer these queries. This process is made efficient by applying set operations, such as intersections and unions, on inverted indexes. However, online computations can still be a very expensive process. In order to solve this problem and reduce the amount of online computation, we introduce the idea of a *Smart View*. These are views computed from attributes of two or more fragments based on user queries. These queries are very dynamic, and therefore a set of views statically selected might quickly become outdated. We therefore create a set of unmaterialized views from incoming queries that cannot be answered by a single fragment. A promising set of views from the unmaterialized views are materialized, depending on the space available, update time, and benefit of materialization. This implies that incoming queries are constantly monitored.

When OLAP users perform data analysis, they usually have their own interest or expectation of the data. Even though they might not have any knowledge about the

Figure 4.1: A data cube example



data, they are usually interested in only a given subset of the data. This subset of data is what the user is usually interested in, and all user inquires and analysis are based on this subset. Any information outside this subset becomes superfluous to the user. In this work, we assume that user interest is related to location. Therefore, as the user travels their interest might also change. We call the subset of data the user *Interest*. In order to identify the subset of data that is of most interest to a user, we constantly monitor incoming user queries and filters, materializing the query results based on a cost model and query frequency. The attributes together with the filter information is added to the Directory Index statistics. The Directory Index is an index structure maintained to hold necessary information including statistics required to create, maintain and search for dynamic views. If the frequency is above a given threshold and cost with respect to the cuboid from which it is computed is minimized, then the query result is materialized for the given user. The cost model is based on performance threshold f. That is, only cuboids whose size is less than or equal M/f are selected for materialization. (Recall that M is the size of the cuboid from which the view is computed where f is threshold value.) Based on the above discussion, the Smart Cube algorithm can be summarized as follows.

The first part of the algorithm utilizes the benefit of shell fragmentation to reduce the storage complexity of the data cube. The algorithm starts by adding a surrogate key (tid) to the fact table to serve as a tuple identifier. The *tid* is used as row or tuple identifier for constructing inverted indexes and for set operations such as intersection and union, utilized during online computation. Next, we flatten the multidimensional star schema into a single flat view to allow for the application of a vertical fragmentation function. The problem of vertical fragmentation in multidimensional databases has been studied

Algorithm 3 The Smart Cube Construction Algorithm

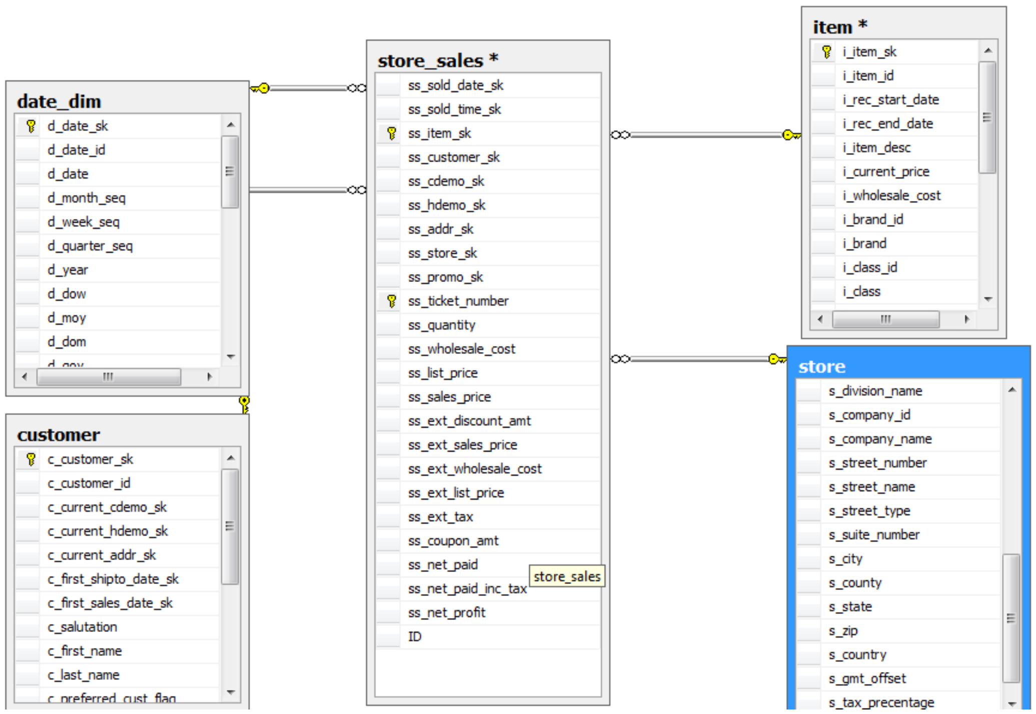
```
Input: Multidimensional Relation R
       Number of Partitions N,
       Smart Size Threshold S,
Output: A set of fragment partitions $\{P_1, \cdots, P_k\}$

 1: Add a surrogate key tid to the fact table;
 2: Flatten the fact table and dimension table into a multidimensional view R;
 3: Partition R into fragments $\{P_1, \cdots, P_k\}$;
 4: Scan fact table once and do the following
 5: Insert each (tid, measure) into $ID\_measure$ array
 6: For each Dimension row in each partition
 7: Build an Inverted Index entry (rowvalue, tidlist) or a bitmap entry
 8: foreach fragment Partition $P_i$;
 9: Build local fragment cube by materializing the most beneficial cuboid based on the cost model and
    Row Threshold using Smart_Materialization
10: Propagate bitmap index or inverted indexes to all materialized views using set intersection;
11: Continuously build Smart Views are monitoring in coming queries
12: Personalized the resultant cuboids using Smart_Personalization
```

in [43], but the authors proposed no new algorithm for determining an optimal fragmentation. However, [20] proposes an algorithm for determining an optimal fragmentation. Following this idea, we fragmented our multidimensional database by selecting dimension attributes that minimizes the total number of rows. This is achieved by selecting all the partition attributes from one or two dimension tables that reduces data sparsity. To this effect, we computed the cardinalities all the attributes in the dimension space. The cardinality ranges that forms a single fragment are then specified, for example we may ensures that smaller cardinalities form a single fragment and bigger cardinalities also forms their own fragments. Smaller cardinalities ensures that aggregations based on those attributes are small, thus creating smaller fragments. Another simple approach is to compute a fragment from one dimension table, since most dimension tables have relatively smaller number of rows and hence aggregation over them will result in a small fragment size.

Figure 4.2 shows an example of a multidimensional star schema. For example, if our objective is to partition such a schema, we can create a partition by selecting all the partition attributes from a single dimension table, together with measure attributes to form a fragment. In some situations, because the data are temporal, we can add the date and time dimension attributes to each fragment. A resultant fragment view will be made up of the following attributes, *i_item_id, i_item_desc, i_current_price, i_brand,*

Figure 4.2: A Multidimensional Relation



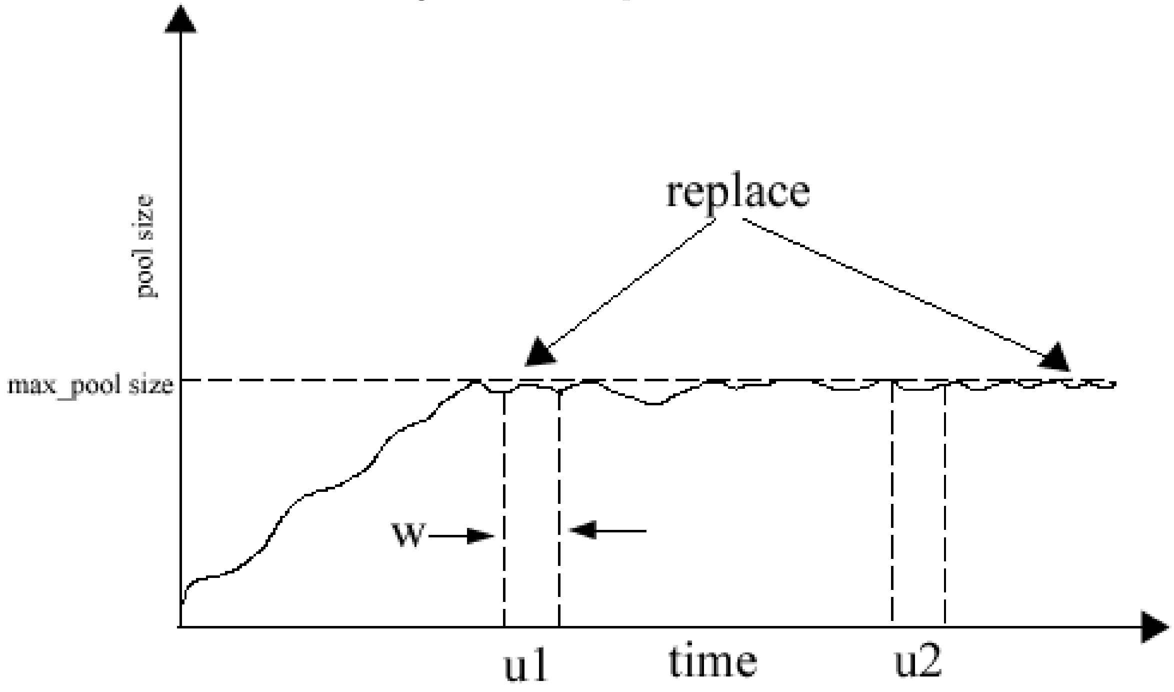
i_class from the item dimension and *ss_coupon_amt, ss_sales_price, ss_list_price* from the *sales_store* fact table. If we add temporal information to each fragment view then the following attributes will make up a fragment, *s_store_id, s_store_name, s_city, s_country, s_county, s_state, d_date, d_month, d_year, ss_coupon_amt*, and *ss_sales_price*. Vertical partitioning of dimension space can reduce the storage complexity of the resulting data cube drastically. To illustrate this, let us consider the data cube in Figure 4.1. A data cube computed from dimensions A, B, C and D will generate $2^4 = 16$ cuboids. However, assume that we partition it into two, such that, we have the first fragment data cube with dimensions A and B and second fragment data cube with dimensions C and D. In this case, the storage complexity of computing the two cubes is $2^2 + 2^2 = 8$. Before computation of the local fragment cube, we build an Inverted index or Bitmap index for each dimension row. Recall from Chapter 2 that Bitmap indexing has been successful to improve the efficiency of important query classes involving selection predicates and joins. It is most efficient when domain cardinality is low. Inverted indexes, on the other hand is advantageous especially in high cardinality attribute domain. Both Inverted indexes and Bitmap indexes facilitate easy online query computation of fragment cubes using set operations.

The computation of local fragment cubes are build using our proposed Smart Materialization algorithm, as follows. The computation of a localized cube for each fragment is done by selecting only the most beneficial cuboids for materialization and then build a Bitmap or Inverted index for each selected cuboid. Next, we compute the Smart Views by constantly monitoring and materializing incoming user queries to reduce online cube computations. Smart views are cuboids computed from two or more fragment views. The Smart Views are materialized based on the frequency of queries that are answered using that view and also the benefit of materializing the view. Finally, we personalize the Smart Cube by using the Smart Personalization algorithm. This algorithm takes user interest as input and continuously monitor user incoming queries and materialize the results for each user.

4.2.1 Smart View Algorithm

When a multidimensional view is partitioned into fragments and local cubes are computed for each fragment, user queries are answered either from a single fragment or multiple fragments. This implies that answering certain user queries requires joining multiple fragment views using Inverted indexes to facilitate online computation of cuboids. In order to reduce the number of online query computation, views capable to answering queries that require on-line computation are created. These views are materialized every time a user poses a query to the system which cannot be answered by existing views or which the cost of answering the query using existing view is high. The most beneficial set of views that are materialized are, however, done under the constraint of S; the storage space allocated for these views. These materialized views are referred to as *Smart Views*. The Smart Views cannot be statically selected because user queries are very dynamic. In view of this, the Smart View algorithm extends the DynaMat Algorithm as introduced in Section 3.2. It constantly monitors incoming queries and materializing a promising set of views, subject to space and update time constraints. When a user poses a query, the system uses a *view locator* to first check if it can be answered from a single fragment. If not, the algorithm then checks within the existing Smart Views to determine whether it can be efficiently used to answer the query. If no view is available to answer the user query or if the cost of answering the query using existing view is too high, it then uses Inverted indexes to compute the query online from multiple fragments. The resultant view may or may not be materialized. This decision is based upon a cost model that compares the cost of answering a query through the Smart View set V with the cost of answering same query against the unmaterialized view computed from the intersection of Inverted indexes. During this phase, the goal of our computation algorithm is to reduce on-line computation through the Inverted indexes and to answer as many queries from the Smart Views as possible. At the same time, the algorithm will adapt to new query

Figure 4.3: The space bound



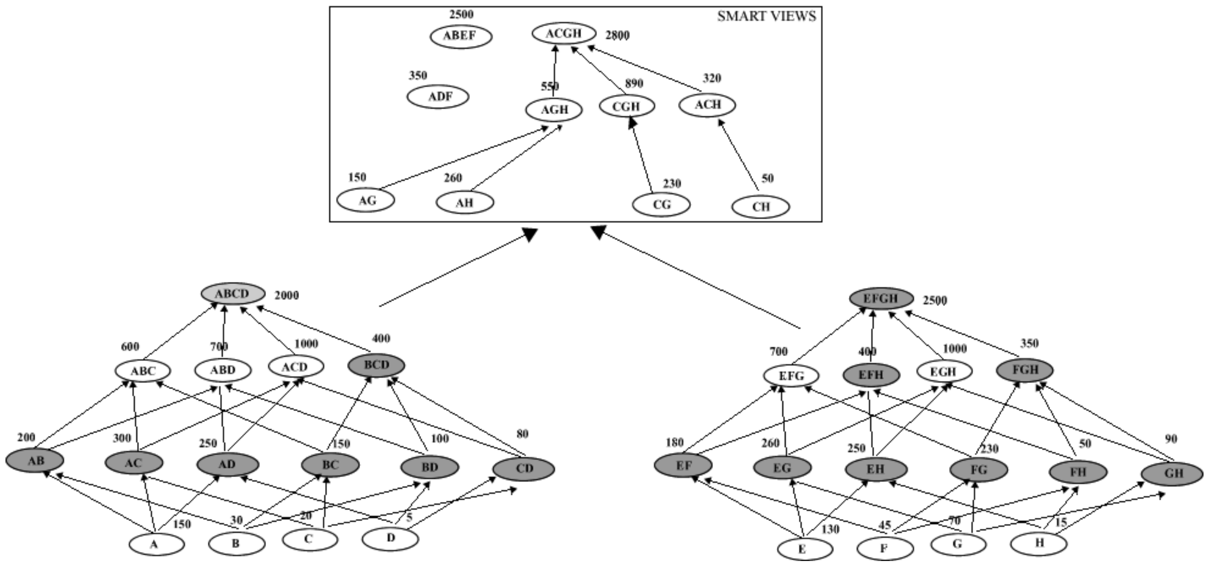
patterns by creating new views and removing less frequently used views. When a user query is computed from on-line computation, a view based on the query is created and added to the pool. This, however, does not add to the storage space, since the view is not materialized until it is deemed profitable to materialize. A promising view for materialization must meet the following criteria:

- The number of queries answered using that view must meets a frequency threshold
- The cost of materialization minimizes the overall query cost

If the smart pool had unlimited storage space, the size of the materialized data would grow monotonically over time. However, a *space bound* threshold on the system serves as a constraining factor. In the case where the pool becomes full, a *replacement policy* is applied to make space for new views. This can vary from a simple naive approach of not admitting more materialized views to the smart pool, to known techniques such as *Least Recently Used* (LRU) and *First In First Out* (FIFO), amongst others. Figure 4.3 depicts a Smart View pool with storage as the limiting factor that calls the replacement policy.

The replacement policy is based on a *goodness measure* for choosing which of the Smart Views we prefer. Each time the space limit for the Smart Views is reached we use

Figure 4.4: A sample Smart View



the *goodness measure* to determine whether to replace the views or not. The following are are goodness measures we used.

- The time that the view was last accessed by the system to handle a query

$$goodness(f) = t_{last\_access(f)} \tag{4.1}$$

  This information is kept in the Directory Index, which we will discuss later. Using this time-stamp as a goodness measure, results in an Least Recently Used (LRU) type of replacement in both cases.

- The frequency of access $freq(v)$ for the fragment

$$goodness(f) = freq(v) \tag{4.2}$$

  The frequency is computed using the statistics kept in the Directory Index and result in a Least Frequently used (LFU) replacement policy

Figure 4.4, shows a sample Smart Cube made up of two fragment cubes and Smart Views computed from fragment ABCD and EFGH. Each fragment is partially materialized with the views shaded in gray. For example, AGH is a view computed using dimension attribute A from fragment ABCD and dimension attributes GH from fragment EFGH. Notice that, although, a query with attributes AGH can be computed from ACGH, the cost of computing it from AGH is less than the threshold value M/f. Assume the limit of the view pool storage for the Smart View is reached. Then, any extra view that meets the frequency and cost threshold will have to replace an existing view or views. In such a situation, we check the number of times each view has been used to answer a query and sort them in ascending order. Subsequently, the view with the least frequency is removed. The size available is checked to see if there is enough space to add the new view. If not, the next least frequently used view is then removed. This process continues until there is enough space for the new views to be added to the pool.

4.2.2 Smart Materialization Algorithm

After partitioning the multidimensional view, a local data cube is computed for each fragment using a partial materialization algorithm. Next, we introduce our partial materialization algorithm, the *Smart Materialization* algorithm that aims to reduce the number of cuboids materialized for each fragment. The aggregated base table for each fragment is the base cuboid c_b. Our algorithm reduces the number of views that are materialized by eliminating views whose size is less than Row Threshold η and selecting only relatively small views that guarantees fast query response time. Given f an input parameter called the performance threshold, a view is considered *small* if its cost is less than M/f, where $M = size(c_b)$. Keeping small cuboids for materialization guarantees a certain quality when querying the data cube. To explain this further, consider the data cube example in Figure 4.1. It represents the data cube lattice obtained from a fact table T whose dimensions are A, B, C and D. Each node is a cuboid and is labeled with its dimensions together with its size. There is an edge from c_1 to c_2 if and only if c_1 can be computed from c_2, $c_1 \neq c_2$ and there is no c_3 such that $c_3 \neq c_1, c_3 \neq c_2, c_3$ can be computed from c_2 and c_1 can be computed from c_3. The topmost cuboid is the base cuboid and corresponds to the fact table. The minimal cost of evaluating all queries corresponds to the case where each cuboid is precomputed and stored. Thus, for this example $minCost = \sum_{i=1}^{2^4} size(c_i) = 5985$. In contrast, the maximal cost corresponds to the situation where only the base cuboid is stored. In this case, every query is computed from ABCD and thus has a cost proportional to the base cuboid size. Hence, $maxCost = 2^4 \times size(ABCD) = 16 \times 2000 = 32000$. This is the minimal amount of memory required to answer all queries. Assume that $S = ABCD, AC$. The performance measures of S are as follows: The memory required to store S is $Mem(S) = size(ABCD) + size(AC) = 2000 + 300 = 2300$. The cost for evaluating all the 2^4 possible queries is computed as shown: First, consider the stored cuboid AC; this can be used to answer queries A, C, AC and *apex*. All these queries can, also, be computed from ABCD. However, using this to compute say A will require more time than using AC. Thus, the cost of S corresponds to the sum of costs of evaluating AC, A, C and *apex* from the cuboid AC and all other queries from ABCD. Hence $Cost(S) = 4 \times size(AC) + 12 \times size(ABCD) = 25200$.

Let us now consider the cuboid AC and BC. Their respective performance factors with respect to S are $f(AC, S) = \frac{cost(AC,S)}{size(AC)} = 300/300 = 1$ and $f(BC, S) = \frac{cost(BC,S)}{BC} = size(ABCD)/150 = 2000/150 \approx 13$. This means that by storing ABCD and AC, the cost of evaluating the query AC is exactly the minimal cost, but for evaluating the query BC the cost is approximately 13 times the minimal one. However, although the performance factor gives an approximation of the cost of executing queries, it does not give an accurate picture of the query response time. This is because the time taken to execute queries by most DBMS after a certain number to tuples remains the same even if the number of tuples is further reduced [31][48]. For example, the time required to execute a query on a 100 rows table might be the same as the time required to execute the same query on a 20 row table, because they may involve the same number of I/Os. Therefore, if we estimate this threshold we can reduce the search space significantly

> *Performance Factor*: Let S be the set of materialized cuboids and c be a cuboid of C. The performance factor of S with respect to c is defined by $f(c, S) = \frac{cost(c,S)}{size(c)}$. The average performance factor of S with respect to $C' \subseteq C$ is defined by $\tilde{f}(C', S) = \frac{\sum_{c \in C'} f(c,S)}{|C'|}$.

Intuitively, the performance factor measures the ratio between the response time for evaluating a query using S, a given materialized sample over the query time when the whole data cube is materialized. This implies that the minimal cost to evaluate a query c corresponds to size(c). When c is materialized, we reach the minimal cost. However, if c is not materialized, it is evaluated using one of its ancestors present in S. The performance factor therefore measures how far the time to answer c is, from the minimal time.

The algorithm starts by setting the set of materialized views to a Null set. We then start from the base cuboid and list all the children of the base cuboid. The size of each child node or cuboid is computed. If the size is less than M/f, it is added to the set of Materialized cuboids. Next, we iterate through all the nodes generated and for each node we generate a list of child nodes called candidates. The size of each cuboid generated is computed and compared against M/f. If the size of a cuboid is less than this threshold and the size of its smallest parent is less than the row threshold, then the cuboid is added to the set of materialized cuboids. To show a simple example, the fragment cube ABCD in Figure 4.4 is computed using performance factor $f = 5$ and Row threshold $\eta = 300$. The shaded ones for example BCD, is materialized because 2000/5=400. AD is also materialized. However, AD meets two criteria before it is materialized. Thus $250 < 2000/50 = 400$ and 2000 is also greater than 300.

Algorithm 4 The Smart Materialization Algorithm

```
Input: Partition P
       Number of Partitions N,
       Parameter f,
       Row Threshold $\eta$,
Output: $S$ Set of materialized cuboids

1: for each P in parttions
2: Initialize the set of materialized views to an empty set $\phi$
3: Initialize $C_1$ to the set of all cuboids under base cuboid thus all children of $c_b$
4: Initialize $L_1$ to the set of all cuboids $c$ in $C_1$ such that $|c| \leq M/f$
5: $S = S \cup L_1$
6: for $i = 1; L_i \neq \phi; i++$ do
7:     $C_{i+1}$={$candidates\ generated\ from\ L_i$}
8:     for each $c \in C_{i+1} do\ compute\_size(c)$
9:     $L_{i+1} = c \in C_{i+1} s.t|c| \leq M/f$
10:    for each $c \in L_i$ do
11:    if $c' > \eta$ s.t $c' \preceq c$ Then
12:    $S = S \cup \{c\}$
13: end for
14: Return S
```

The procedure *compute_size* is implemented either by calculating the actual size of the cuboid argument or by using estimating size techniques such as those discussed in [4]. The cuboid size estimation technique is preferred when we want to reduce computation time. The maximal complexity is less than 2^D, since the algorithm does not go further down the cube lattice when the row number threshold is reached. Indeed, even when $f = 1$ and all cuboids have size less than M/f, not all cuboids will be computed unless the minimum row number threshold is 1. Of course, in practice the minimum row number threshold is far greater than 1 and $f > 1$, therefore the actual complexity is much less than 2^D. The interesting thing here is that the algorithm is run for each partition, therefore reducing the overall complexity to much less than 2^D.

4.3 Personalization of Smart Cubes

Recall that, in most very large data warehouses, users are mostly interested in only a subset of the data and most data analysis is done within this subset. For example, a manager in Ottawa might prefer to perform all his cube analysis within the Ottawa data subset. That is, this manager might want to view the Ottawa data subset as a whole cube within which all OLAP operations can be encapsulated. In order to meet this user need, we have to personalize the data cube for individual users. Since users' interests usually arise in an impromptu way and usually changes over time, we cannot precompute this personal cube for each user. Instead, we dynamically construct the cube by monitoring user queries and materialize the results based on a cost model. Since space is limited, we should not suppose that users have unlimited storage space available. In fact, usually there are many users of the OLAP system, and different users have different levels of priority. In this section, we suppose that a user constructs his Personalized Cube under the constraint Ω of total available storage space.

First, the system continually monitors incoming user queries, taking into account the frequency of the query and computing the profit of materializing the query. Using a weighting method based on filter factor and frequency of query, we accept queries for materialization in the personalize pool. Each time a new query is selected for materialization, we check for available space. If space exist, we materialize, if not we replace an outdated query. Note that only queries filtered on user interest are considered for materialization.

> *User Interest*: A user interest I of R is a filter over dimension levels. $I = p_1 \vee p_2 \vee \cdots \vee p_k$, where $p_i = f_{(i,1)} \vee f_{(i,2)} \vee \cdots \vee f_{(i,k)}$, k is the number of attributes of R and $f_{(i,j)}$ is a predicate about the j^{th} attribute of R. Let R'=$t|t \in$ R and t meets I and I_D=$A_j|A_j$ is an attribute, and $\exists p_i = f_{(i,1)} \vee \; f_{(i,2)} \vee \cdots \vee f_{(i,k)}$, where $f_{(i,j)}$ is not a tautology, we call I_D *interesting attribute set*, and A_j is an interesting attribute.

The user interest I presented in this section is based on location. Thus, when a user moves around his interest changes based on his current location. However, Interest can also be based other attributes such as customer class, and product category, amongst others.

> *Filter factor*: Suppose that the total number of the dimension elements is α, and there are only β elements that meets the user interest I. Then the filter factor of I is $\delta = \frac{\beta}{\alpha}$.

The algorithm as depicted above, is very similar to the one we saw for constructing Smart View in Section 4.2. However, the cost of selecting a view to be materialized is determined by the filter factor. The filter factor is the percentage reduction in size between query result and the cuboid used in answering the query. The algorithm continuously monitors a user's incoming queries. The algorithm then adds the interest to the filter of the query if its not already in the query. This ensures that all cube operations are done

Algorithm 5 Data Cube Personalization Algorithm

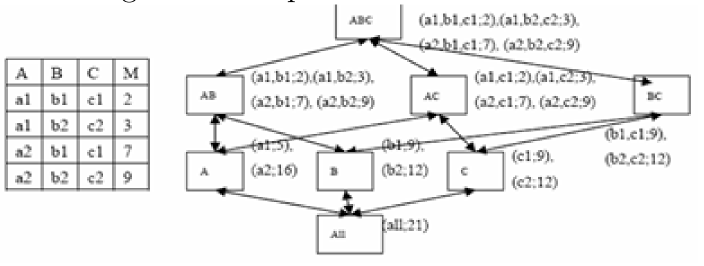
```
Input: User Query q
       Cuboids C,
       Storage Space Threshold $\Omega$,
       Accepted Thresholdl $T$,

 1: SET pool size =$\Omega$
 2: Initialize personalize pool to $\phi$
 3: Monitor incoming query $q$
 4: for each query $q$
 5: Add I to conditions of q if q does not already contain I
 6: $t = Compute\_size(q)$
 7: Compute the filter factor $\gamma$
 8: Compute the frequency $f$
 9: if $\Omega > 0$ AND $f + \gamma > T$
10: $\Omega = \Omega - t$
11: Materialize q
12: $P = P \cup q$
13: Update the accepted threshold based on existing query frequecy
14: else if($\Omega = 0$ AND $f + \gamma > T$) call replacement policy
```

under the constraint of user Interest. The size of the query result is then computed. This result size is used to compute the filter factor. The frequency of the resultant query is also computed by incrementing, by 1, any time the same query is issued. Next, we check to see if the storage space threshold has not been exceeded, i.e. if there is enough space to store the resultant query. We also compare the frequency and filter factor of the query result with their respective thresholds. The filter factor is kept constant. However, the frequency threshold might be initially set to a very low value, to allow all queries that meet user interest as long as there is space to be added to the pool. As more queries are materialized the frequency threshold is increased by using either the average threshold, or median threshold or the modal threshold of existing materialized views. If the space threshold reaches the limit, the replacement policy is used to replace new query results with existing ones. The replacement policy used for Smart Views is used to handle new queries that meet the requirement for addition into the pool.

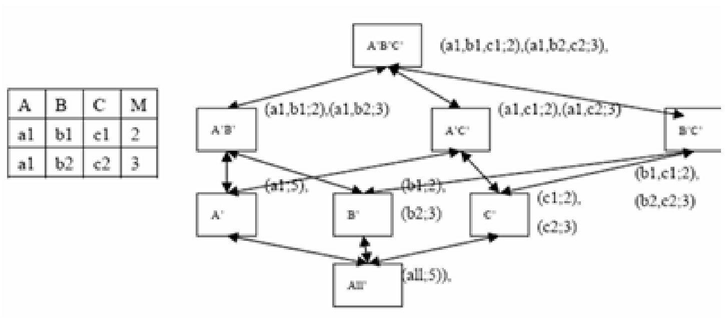
So far, we have presented a partial materialization algorithm that reduces the space and time complexity of the data cube computation. We have also personalized the cube to ensure users get only the most interesting subset of data. The algorithms presented, materializes views at different levels with the aim of reducing storage space while improving query processing time. For example, a personalized cube can be computed on the client while the Smart Cube is computed on the server. The challenge here is how

Figure 4.5: A personalized data cube



A	B	C	M
a1	b1	c1	2
a1	b2	c2	3
a2	b1	c1	7
a2	b2	c2	9

A: Lattice of original cube



A	B	C	M
a1	b1	c1	2
a1	b2	c2	3

B: Lattice of Personalized cube

user queries will be answered by choosing the right cuboid for optimal performance.

Suppose a dataset R shown in Figure 4.5 has three dimensions: A, B, C and it has one measure M. Assume the user interest I='a1'. Figure 4.5B shows the personalized cube lattice and all the cells of this cube. The cuboid pair $< (AB), (A'B') >$ is a dataset pair where AB is from the base cube lattice shown in Figure 4.5A and A'B' is shown in Figure 4.5B. Note that, in the example in Figure 4.5, the data cube is assumed to be fully materialized.

4.4 Querying the Personalized Smart Cube

When a query q is posted to the system, we first scan the personalized cube to answer q. Given a personalized pool P and a query q, P answers q if and only if for every query posed to the pool, the exact same query is stored in the pool or the stored query's dimensions spans the whole domain of dimensions of the posed query. If no stored query is found to answer the posed query, the Smart Cube is used to answer the query. In order to simplify the process two Directory Indexes are kept, one for the personalized pool and the other for the Smart Cube. The Directory Index is a set of indexes connected through the lattice, as was shown in Figure 3.1. Each node has a dedicated index that is used to keep track of all materialized views in the partition fragments and the views in the Smart

View pool. However, if the Directory Index corresponds to a personalized cube, then it keeps track of all materialized queries and their corresponding views that are stored in the pool.

> *OLAP Query*: An OLAP query usually has the following format Q(*measure*, *cuboid*, *condition*), where *measure* is aggregated value of a cell, *cuboid* is representing requested dataset, and *condition* is restrictive predict over this dataset with filter factor t.

Algorithm 6 Query Answering Algorithm

```
Input: query q
       Directory Indexes P and M
       Main Cube C
       Personalize Cube $C'$,
Output: $R$ Query Response

1: Using Directory Index P search if query can be answered using personalize cube
2: if a view fragment is found search within the exact query fragment to respond $\phi$
3: If query q cannot be answered by P Search the Directore Index M
4: Using M identify the right fragment thus where partition or Smart View section
5: Select the cuboid with the least cost that can answer the query.
6: Return query response;
```

The Directory Index P is the Directory Index for the personalized cube, while, M is the Directory Index for the Smart Cube. The difference between the two Directory Indexes is that M is sub-divided into two sections. The first section keeps track of fragments and their cubes while the second section keeps track of the Smart Views. The Directory Index for personalized cube, on the other hand, has only one section. This Directory Index is made up of fragments r. For each fragment r of the Directory Index there is exactly one entry that contains the following information:

- Hyper-plane $\vec{r}$ of the fragment
- Statistics (e.g. number of accesses, time of creation, last access)
- The father or parent of r similar to parent of a cuboid in a lattice

For our implementation we used R-trees similar to the DynaMat algorithm, based on the $\vec{r}$ hyper-planes to implement these indexes. When a query q arrives, we scan using all views $\vec{q}$ in the lattice, that might contain materialized queries r whose hyper-planes $\vec{r}$ covers $\vec{q}$. For example, if $\vec{q} = (1, 1000), (), Smith$ is the query hyper-plane for dimensions *product*, *store* and *customer*, then we first scan the R-tree index for view (*product*, *customer*) using rectangle (1, 1000), (Smith, Smith). If no cuboid is found, based on the dependencies defined in the lattice, we also consider view *(product, store, customer)* for candidate cuboid. For this view, we "expand" the undefined in the *store* dimension and search the corresponding R-tree using rectangle $\{(1, 1000)$, $(min_{store}, max_{store})$,$(Smith, Smith)\}$. If a fragment is found, we "collapse" the *store* column and aggregate the measure(s) to compute the answer for q. Based on the content of the pool or content of the materialized views, there are three possibilities. The first is that, a stored query or view matches exactly the definition of the query. In this case result R is retrieved to the user. If no exact match exists, assuming we are given a cost model for querying the fragment, we select the parent or ancestor with the least cost from the pool, to compute q. If no fragment can answer q, then control is sent to the second Directory Index to search a cuboid within the Smart Cube to answer the query. The search within the two Directory Indexes is hierarchical in nature. If no parent is found to handle the query within the Directory Index for personalize cube, control is transfered to the Directory Index of the Smart Cube.

4.5 Conclusion

In this chapter, the details of our Personalized Smart Cube approach were provided. The proposed algorithms have the potential to ensure high query performance, by systematically adding and materializing views that keeps the overall data cube size small. Our approach further dynamically personalizes the data cube for individual users. The personalized Smart Cube approach includes the Smart Cube algorithm and the Personalization algorithm. First, we discussed the Smart Cube construction algorithm, where we partitioned the cube into fragments and joined using Inverted Indexes. Next, we reduced the rate of online computation by introducing the Smart View algorithm, that dynamically materializes views based on user incoming queries. We also discussed our materialization algorithm, that significantly reduces the computation complexity by using a Row Threshold. The Smart Cube computed from our algorithm so far produced generalized results. However, users are mostly interested in only a subset of the data that is of interest to them. To provide a solution to this problem we introduced the Personalized Smart Cube algorithm. Finally, we discussed our query answering algorithm. In the next chapter we discuss how we evaluated our algorithm. We discuss how the system was designed and the databases that were used.

Chapter 5

Experimental Design

In this thesis we introduced a novel Personalized Smart Cube approach for data cube computation and personalization. In Chapter 4, we detailed the algorithms required in computing the data cube, personalizing the data cube, and answering queries. In this chapter we describe the databases that were used for implementing the proposed solution. The implementation steps for the solution are presented, together with the definition of the experiments performed.

First, in Section 5.1 we give a general description of the databases that are used in our implementation. We describe the TPC-DS database, a synthetic database and the US Census database. In Section 5.3 we briefly explain our implementation, as well as the architecture of our proposed system. Specifically the implementation of the partitioning and materialization algorithms are explained. Using activity diagrams, we depict the major activities of the Smart Cube technique. Next, we show how we implemented the data cube personalization, and we explore the impact of different storage allocations on the personal cube on the system. The implementation of our query processing algorithm is also detailed and the activity diagram of the process is presented. We examine the various processes and the part they play within a query process task. Finally a summary is presented in Section 5.4.

5.1 Experimental Databases

In this section, we describe the databases that were used in our experiments.

5.1.1 TPC-DS Database

TCP-DS is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance [58]. Although the underlying business model of TPC-DS is a retail product supplier, the database schema, data population, queries, data maintenance model, and implementation rules have been designed to be broadly representative of modern decision support systems. This data mart contains nine (9) dimension table schema with each of their key constraints referentially connected to a fact table schema. The combined dimension attributes in the dimension tables is 80 and the number of measure attributes in the fact table is 10. This implies that a full cube materialization of the data mart will be infeasible due to the 2^{80} lattices to store. The sales fact table schema contains 3,000,000 tuples.

Figure 5.1 shows part of the star schema representing our application, which is derived from the TPC-DS star schema benchmark [58]. The *store_sales* is the fact table that measures sales and orders, while *customer, store, promotion* and *date_dim* are the dimension tables that store descriptive attributes that categorized the facts. These dimension tables are referenced by the fact table through foreign keys. In addition, the dimension tables hold hierarchies that enable data aggregation according to different granularity levels, such as $(s\_country) \preceq (s\_state) \preceq (s\_city) \preceq (s\_address)$, which is held by the dimension table *store*, and $(i\_manufact) \preceq (i\_category) \preceq (i\_subcat) \preceq (i\_item_sk)$ which is held in the dimension table *item*. Considering the mentioned hierarchy in the dimension table Store, *s_country* is the highest granularity level, which *s_address* is the lowest granularity level. According to [30], $Q_1 \preceq Q_2$ if and only if it is possible to answer Q_1 using just the result of Q_2, and $Q_1 \neq Q_2$. Therefore, it is e.g possible to find out the revenue in a given country by aggregating the results of the cities inside that country.

5.1.2 US Census 1990 Database

This database is a multivariate dataset with 2,450,000 tuples and 68 categorical attributes [6]. It is an open source database located at (`http://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)`). It was collected as part of the 1990 US census. The continuous variables have been discretized and the discrete variables that have a large number of possible values have been collapsed to have fewer possible values. The following are the attributes of the database *age, ancstry1, ancstry2, Avail, Citizen, Class, Depart, Disabl1, Disabl2, English, Feb55, Fertil, Hispanic, Hour89, Hours, Immigr, Income1, Income2, Income3, Income4, Income5, Income6, Income7, Income8, Industry, Korean, Lang1, Looking, Marital, May75880, Means, Military, Mobility, Mo-*

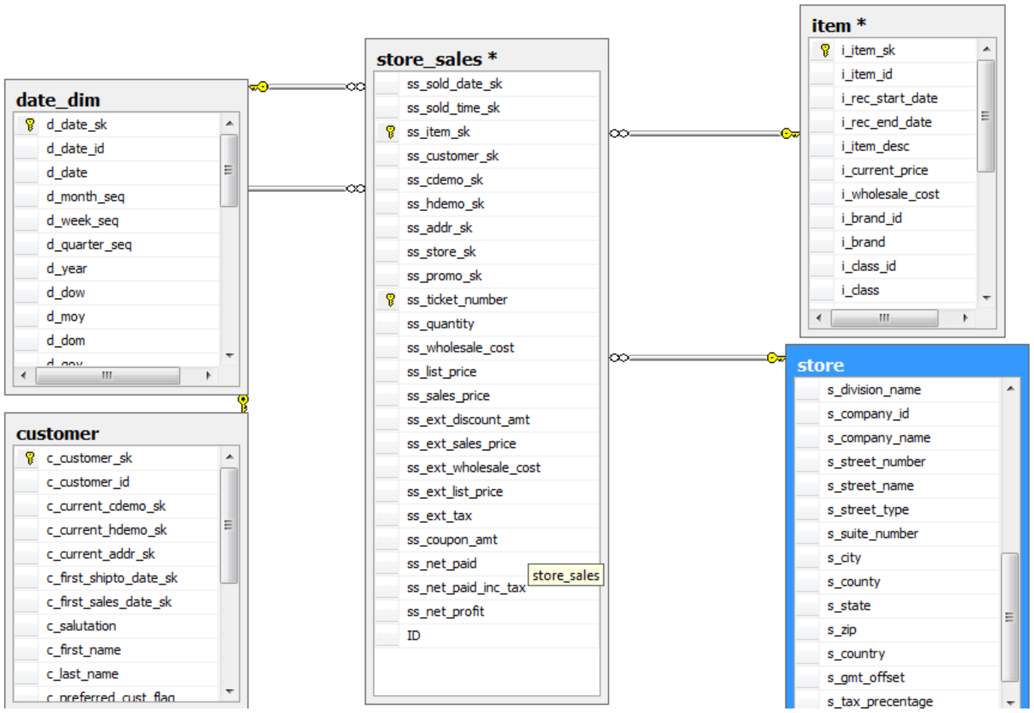


Figure 5.1: A star schema for a DWH of a retail application

billim, *Occup*, *Ohterserv*, *Perscar*, *POB*, *Poverty*, *Pwgt1*, *Ragechld*, *Rearning*, *Relat1*, *Relat2*, *Remplpar*, *Riders*, *Rlabor*, *Rowchild*, *Rspouse*, *Rvetserv*, *School*, *Sept80*, *Sex*, *Subfam1*, *Subfam2*, *Tmpabsnt*, *Travtime*, *Vietnam*, *Week89*, *Work89*, *Worklwk*, *WWII*, *Yearsch*, *Yearwkk*, and *Yrsserv*.

5.1.3 Synthetic Database

This database contains 10^6 rows and 20 dimensions. Many observations have shown that the attribute values of real dataset do not follow, in general, a uniform distribution. Rather, they often follow a power law distribution [29]. That is, if we sort the values of a given attribute in the decreasing order, then the frequency of the value of rank i is proportional to $\frac{1}{i^a}$, $a > 0$. a belongs mostly to the range [2,3]. Based on this observation, this database is used in evaluating our algorithm follows the Zipf distribution and varying Zipf factors. A Zipf factor of 0 means that the data is uniformly distributed. By increasing the Zipf factor, we increase the skew in the distribution of distinct values in the datasets. The Zipf factor used ranges from 0.0 to 3.0 with 0.5 intervals; these were set by inspection.

5.2 Experimental Setup

All algorithms are implemented in C# and all the experiments are conducted on an Intel I5 generation processor 2.30 GHZ with 6GB of RAM. The system runs a Windows 7 operating system. We tested our approach on the SQL Server Database, using the well known TPC-DS benchmark, Synthetic dataset and US Census 1990 dataset as previously introduced. For each database, before data cube computation, the data is first preprocessed. This consists of performing data de-duplication, replacing NULLS with NA and Binning the dimension attributes where applicable. For example, the *age* dimension is best utilized when it is binned into Age Group. After the database has been cleaned, the data source is selected. If the source database is modeled as a star schema, we flatten the tables using the key field to create a single view. The aim of this view is to allow any query to be computed easily from multiple tables. When using the synthetic or the US Census 1990 database, we do not need to flatten, since they are already flat files. We varied the dimensionality and cardinality of the synthetic database by experimentation, so as to evaluation our algorithm based on different database settings such as high dimensionality and very large databases, amongst others. When creating a partition from a database that is modeled as a star-schema, we create the partition view by selecting attributes of interest from one or two dimension tables and measures from the fact table. This view is computed from a join between the dimension table primary key and the fact table foreign key. However, if the dataset is not in the form of a star schema, the attributes of interest are selected and partitioned based on what attributes are most likely to be queried together and also what attributes will create a smaller sized view. As part of evaluating our approach, we will also consider the performance of our system when Smart Views are not used. We will refer to it henceforth as *Smart Cube with no Smart Views.* When no Smart Views are used, it implies that queries which require a join between two fragments are computed using Inverted Indexes. The performance gain when the Smart Views are incorporated for answering queries will also be examined.

Finally, although the main function of the personalized Smart Cube is to provide users with a subset of the data based on their interest, it is also meant to provide an ultra-fast query response to users. We therefore evaluate performance gains by experimentally analyzing them. We randomly generated 60 queries as shown in Appendix A, that is tuned with different statistical properties. In order to ensure that our recorded query times were reproducible, we executed the queries a number of times with the frequencies of some queries being more than one. We used the average time and time elapsed to evaluate the processing time. We also used query profiles in our experiment to evaluate the query processing time, as discussed next.

5.2.1 Query Profile

A query profile defines a sequence of queries randomly chosen from a set of distinct query types [16]. In the sequence, each query type appears at least once, so that the total number of repetitions of each query type in the sequence is equal to the cardinality of the sequence. A query profile is the frequency distribution of query types found in the sequence. For instance, in query profile 10_90, 10% of the query types have 90% of the frequency, i.e. in a sequence composed of 100 queries and for 10 query types, the frequency of one query type is equal 90, while the total frequency of the other types of queries is 10. Query profiles emulate real word query pattern where a set of queries may have frequencies of submission that are greater than, or less than, another set of queries.

5.3 Implementation Details

The most costly method to process queries over a data warehouse is to perform the *star-join*, by joining all tables of the star schema and then perform filters, groupings and sorting. This strategy provides prohibitive query response times, as discussed later. On the other hand, the methods implemented in this section can improve the query processing performance.

In our method, precomputed data is stored into tables after performing some operations. A *vertically fragmented view* [20] maintains the minimum set of columns of the star schema that are necessary to answer a given query. We created our fragmented view from the data mart by using all required columns from a dimension table and measure columns from the fact table. The joins between the Dimension tables and the Fact table were computed when composing the view, by using the key component of both tables. The views are stored with the aim of improving query processing performance, since joins are avoided and only filters and groupings needs to be computed to retrieve the query answer. On the other hand, *materialized views* pre-compute the data warehouse information that can be used to answer queries that are frequently issued. A materialized view is built by creating a table to report pre-computed data from a fact table that was jointed to a dimension table, and whose measures are aggregated. Since the materialized view stores pre-computed aggregated data, processing a query avoids joins and groupings, and drastically reduces the number of rows thus benefiting the filters.

5.3.1 Architecture of proposed system

The architecture of the Personalized Smart Cube system is shown in Fig. 5.2. The figure shows that our approach accesses various indexes to speed up the OLAP drill-down, roll-up, slice-and-dice and pivoting operations. Before indexes are applied, the system improves performance by partitioning, materializing and personalizing the data cube. On the server side, our system operates a back-end relational database and employs FastBit or Inverted indexes. The server side is responsible for building indexes, answering queries over them, appending new rows and performing some online computations. The queries are submitted by the client to the server, and the server uses query optimization techniques to provide the answer rapidly with high performance. On the client side, the user interacts with our system through a desktop application, submitting queries and analyzing multidimensional data that are rendered on cross tables and charts. Whenever a cross table is modified by the user to produce another view, the corresponding chart is refreshed and synchronized with the cross table. Some implemented facilities allows users to select attributes to index and autocomplete the query string to match the proper syntax. Finally, other utilities manipulate internal files to maintain logs, access privileges, configuration parameters, metadata and parsing.

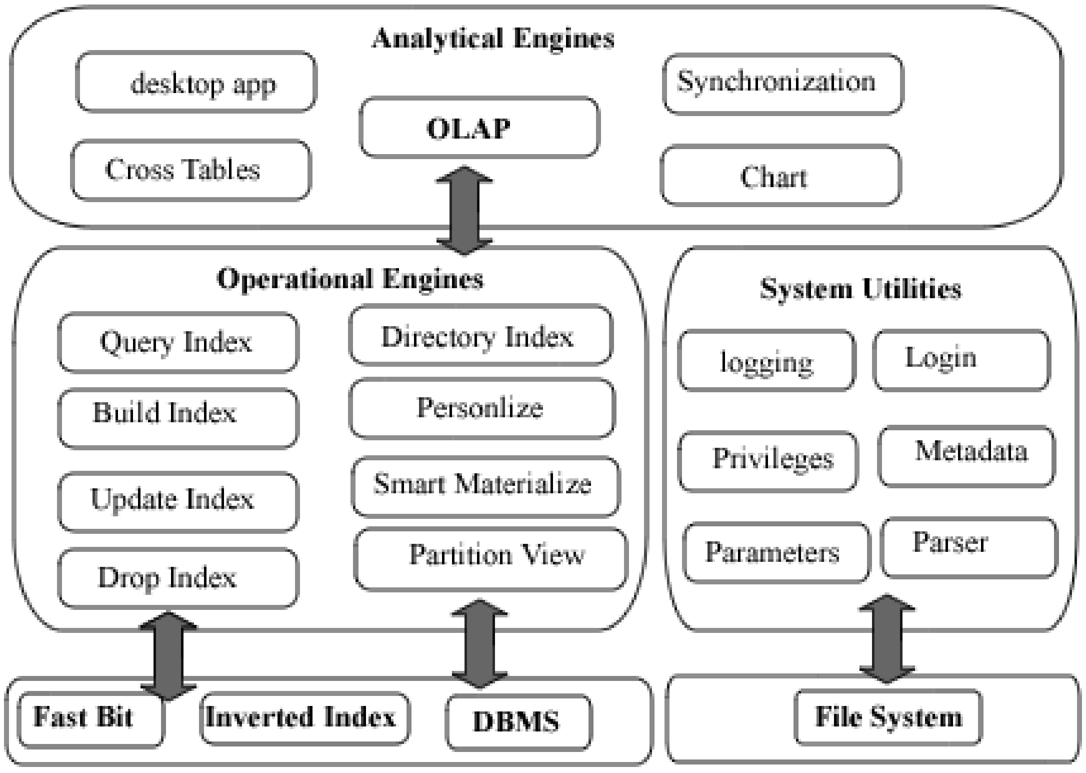


Figure 5.2: The Architecture of the personalized smart cube

The remainder of the subsections discuss the implementation process for view partitioning, materialization, personalization and query processing.

5.3.2 Partitioning and View Materialization

Using the XML editor Mondrian Schema Workbench [1], we specify the dimension and fact tables, the measures, hierarchies that exists in the data warehouse schema and attributes to be indexed. The inputs are validated using the data warehouse schema. If the validation is successful, the XML editor generates a XML document that stores all the data warehouse schema specification and attributes to be indexed. The system uses this information to compute the Smart Cube and build the indexes. Partitioning of the data warehouse is done using SQL scripts to create the views. These views are not materialized, but provide a table-like object to allow for smart materialization of each fragment. The names of these views are passed to the application, which then iterates through each fragment and computes the most beneficial views to materialized. The attributes together with the metadata information for each view are also captured, to allow for easy creation of temporary tables that will be used to materialize the views. After gathering all the necessary information about the views to be materialized, an XML document is parsed by the system. This document issues SQL statements and dump commands or bulk copy commands on the DBMS, in order to compute the joins and create temporary tables using a the appropriate database driver. The DBMS then executes the script, generate the tables and populate it with information.

In order to facilitate efficient storage of the materialized views, Inverted indexes are created for each materialized view. We also Index our materialized views using Bitmap Indexes. The choice of the index used depends on the domain cardinality of the dimension table. Bitmap Indexes are more suited for low cardinality domain, whereas Inverted Indexes are more suited for high cardinality domain. Recall that the aim of these two indexes are to (1) compactly store the data using compression techniques, for quick and easy retrieval of information and (2) allow for online computation of cubes when user query cannot be answered by any of the materialized cuboids. In our implementation we used the FastBit tool for our Bitmap Indexing and PFORDELTA compression scheme for the Inverted Index. FastBit is an open source tool for creating word-aligned hybrid (WAH) [63] compression for bitmaps. If Fastbit is used, the system issues *ardea* and *ibis* commands to the FastBit. While the former reads CSV files to store data into the FastBit binary format, the latter effectively builds the Bitmap Index and stores it into a directory. Finally, the system records metadata that fully specifies the index, e.g. the names and types of the index columns, aliases and the available OLAP operations for that index. The log recording starts after the composition of SQL and dump commands and finishes after metadata are recorded.

Recall that we created our Inverted Index using the PFORDELTA compression scheme. This reduces the space required to store the Inverted Indexes created. Depending on the cardinality of the attribute domain of the dimension table, Bitmap Indexes will be used in place of Inverted Index. (This is because Bitmap Indexes are more suited to low cardinality domain than Inverted Indexes, whereas Inverted Indexes are more suited to high cardinality domain.)

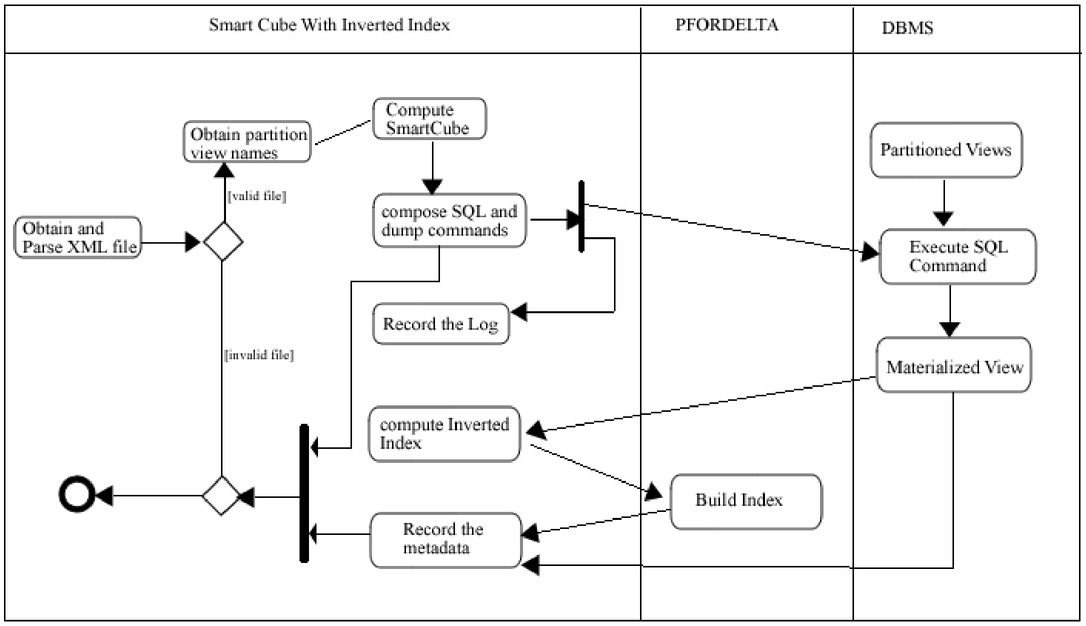


Figure 5.3: Process flow for smart cube with inverted indices

Figure 5.3 and Figure 5.4 shows how the partitioned cubes are created. The name of the partitioned views serves as inputs for creating the materialized views or local fragment cube. The computation of the Smart Cube uses the partition names to compute the local fragment cubes as well as Smart Views. After cube computation, the system composes the required SQL and dump commands for creating all the necessary cuboids and sends it to the DBMS to be executed. The output is a set of materialized views. These materialized views are then exported into CSV files as input to *ardea* and *ibis* to compute (and build Bitmap Index if Fastbit is used). However, if Inverted indexes are used then we do not need to create a CSV file from the materialized views. We only parse the name of the materialized views to the compute Inverted Index command to build the Inverted Indexes.

The idea of using Bitmap Index has the potential of improving query response time

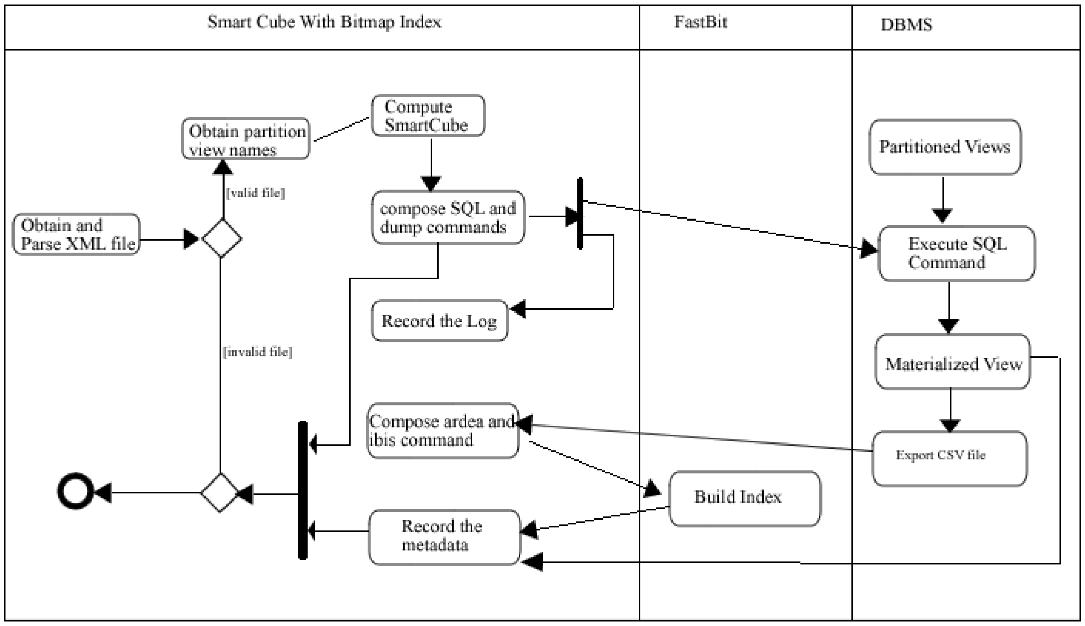


Figure 5.4: Process flow for smart cube with Bitmap index

especially for views computed from a join between multiple tables. For instance, suppose the *c_country, s_country, c_state, s_state, d_year* and *ss_revenue* from our TPC-DS dataset were specified to be indexed. This involves four different tables to be joined, namely *Customer, Store, Date* and ,Store_sales and selecting the appropriate attributes to create a temporary table. The bitmap join indexes are then created on the attributes of this temporary table. Note that Bitmap Indexes are most efficient when applied to such low cardinality materialized views, that are made up of single tables or joins between different tables. It is important to also note that indexing such attributes would enable *roll-up* or *drill-down* operations considering that $(c\_country) \preceq (c\_state)$.

5.3.3 Personalizing the smart cube

In personalizing our Smart Cube, we allocated a fixed size to hold all the materialized views within our DBMS. This means that all personalized views were hosted on the server side of the architecture. Different storage spaces were also allocated to each user, in such a way that the sum of all the sizes allocated to all the users is equal to the total space allocated for personalizing the cubes. Within the application, a Directory Index was created that maps users to their views and contains statistics on how often a materialized views is accessed by the user. The Directory Index also contains information about all the personalized views. Therefore, it is used for searching the appropriate view that will answer user query at the least cost. In our experiments, we use the 3,000,000 tuples of our TPC-DS dataset. We then allocated the following storage space for each user: 512MB, 128MB, 64MB, 32MB, 16MB and 8M. We selected the city called *Fairview* as our filter location. The following were the materialization lists as shown in Table 5.1.

Table 5.1: Effect of storage size on materialized cuboids

Storage	Materialized Cuboids
512MB	All filtered cuboids where materialized
128MB	One filtered cuboid was not materialized
64MB	One filtered cuboid was not materialized
32MB	there filtered cuboids were not materialized
16MB	four filtered cuboids were not materialized
8MB	The same as that when 16MB is allocated

Note that, when a user issues a query, the system will first attempt to answer the user using the personalized cube. If no cuboid matches the user query, the system then transfers control to the Smart Cube dictionary to answer the query. The next subsection explains how queries are answered by the system.

5.3.4 Answering User Queries

When a data cube is built by a user, the metadata is recorded and the cube then becomes available to be queried. The UML activity diagram shown in Figure 5.5 models how to process queries using the system. Initially, the user types in the desired query. The system parses the query and checks within the Directory Index for personalized cube to see if user query can be answered using that personalized cube. If not, the system then uses the Directory Index for the Smart Cube to identify the cuboid that best answers the query.

The system subsequently checks for indexing. If indexes are available, the system writes the proper command containing the query and the chosen index, and submits it to the indexing application. The indexing application accesses the index and processes the query. If no indexing is found, then the system submits the appropriate SQL query and executes it on the DBMS. After processing the query, the system writes a CSV file that contains the query results. The resulting CSV file is read by the system to build cross tables or charts and renders it to the client. After the query execution, users are able to perform various OLAP operations such as *pivoting* by dragging and dropping columns or rows to switch axis of the cross table. *Drill-down* and *roll-up* are also allowed on the requested attributes, if there is at least one hierarchy involved in the previous

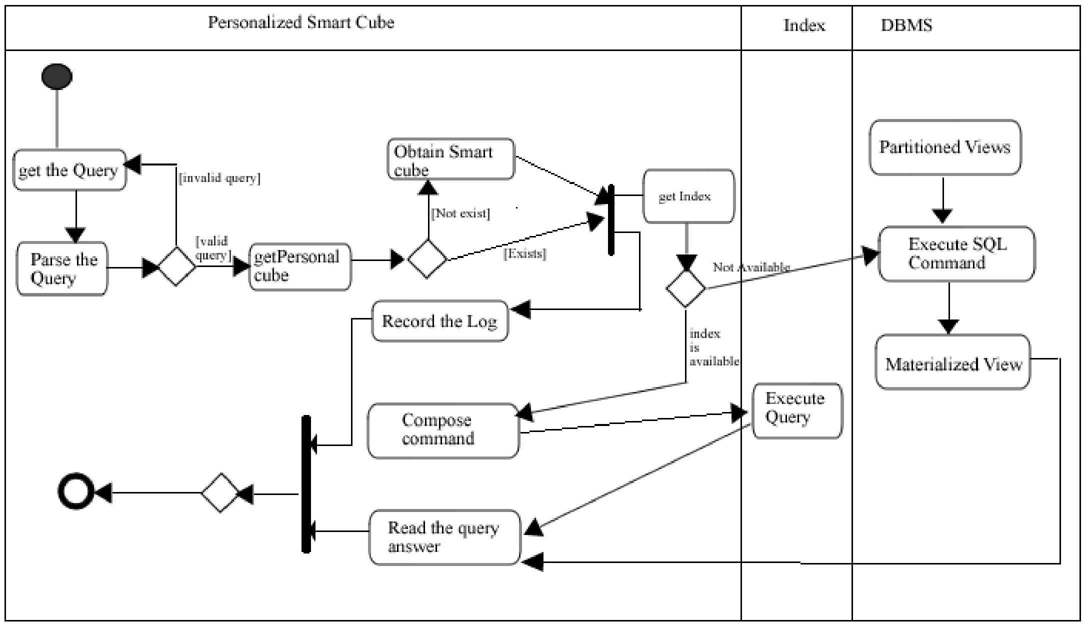


Figure 5.5: Processing queries using the personalized smart cube

query. For instance, if the previous query involves the *s_state* attribute, a combo-box will enable the attribute *s_country* for the *roll-up* operation, and the attributes *s_city* and *s_address* for *drill-down* operation. When Fastbit is used for indexing, the composed queries do not require joins or grouping clauses. The columns listed in the SELECT clause are used to aggregate results. Therefore, writing the query is a straigtforward task for users, since only SELECT-WHERE clauses need to be written. Furthermore, the *slice-and-dice* operation can be described as a restriction in the WHERE clause.

5.4 Summary

In this chapter, the details of the databases used for our experiments were provided. We discussed the dimensions and fact table making up the database, and also discussed the size of the databases, showing that it was appropriate for our intended purpose. As described above in our implementation, the Personalized Smart Cube approach has different components. In the first part, we saw how the DBMS is used to partition the database to develop a smart way of materializing the views and to apply the appropriate index with the aim of reducing the size as well as improving the query performance. Secondly, we showed how the data cube was personalized at the server end using the DBMS and our Directory Index. We also discussed how the materialization list is created, after different storage spaces were allocated to various users. Finally, we discussed how query processing is accomplished within our system. The results from the evaluation of our Personalized Smart Cube approach are described in the next chapter.

Chapter 6

Evaluation and Results

In chapter 4, we presented the Personalized Smart Cube approach, which addresses three (3) of the major shortcomings of current data cube computation approaches. These are reducing the data cube size, ensuring fast query processing time and providing the ability to perform all user cube operations within the data subset that is of interest to a specific user. The results of our experiments are based on the following research questions. Is the overall size of our data cube smaller than the state-of-the-art? What is the relationship between the data cube size and query processing time? Does personalization provide additional query performance?

This chapter provides the results of a comparative study performed on the test databases. It also shows the different types of evaluation methods used to assess the effectiveness of our approach under various circumstances. This evaluation analysis is carried out as a measure in determining and verifying the effectiveness of our proposed methodology.

6.1 Criteria of Evaluation

The performance of our algorithm is evaluate on three main criteria based on the questions asked previously. These criteria are, the overall space consumed by the Personalized Smart Cube, the query processing time and the performance of our personalization algorithm in terms of cost savings. We will analyze the effect of various parameters on these criteria and compare our approach to some state-of-the-art approaches based on these criteria. Next we define the measures used to evaluate our algorithm.

6.1.1 Cost and Memory

In our experiments, the amount of memory is expressed as the number of rows of the materialized views set. Since all materialized views are stored on a storage device, we also refer to the memory space as storage space. Instead of focusing only on the space used to store our materialized views, we also analyzed the time used in computing the data cubes. For the sake of our experiments, to compute the total space used by the smart materialization algorithm, we added all the rows in all the materialized cuboids. This is equal to $\sum_{i=1}^{n} c*_i$ where n is the total number of materialized cuboids. The computation time was derived from our prototype system using the timer module. In order to ensure reproducibility of results, we executed the processes 50 times and report the average execution time.

6.1.2 Performance Factor

Our computation of performance factor is based on the cost model as defined in [29]. Let $S \subseteq C$ be the set of materialized cuboids and v be a cuboid. Then $S_v = w \in S | v \preceq w$ is the set of materialized cuboids from which v can be computed. We define the cost of evaluating a query with respect to a set S as follows. If S does not contain any ancestor of V the $Cost(v, S) = \infty$ otherwise $Cost(v, S) = min_{w \in S_v} Size(w)$. That is, the query is evaluated by using one of its stored ancestors. The chosen ancestor is the one with the fewer tuples. This is the measure usually used to estimate the time complexity [30]. Note that when $v \in S_v$ then $cost(v, S) = size(v)$. This is the most advantageous situation for v. We also define the cost of a set S as the cost of evaluating all queries with respect to S, that is, $cost(S) = \sum_{c \in C} Size(c)$. For our experiments, the performance factor is based on the cost of a query q.

6.1.3 Cost Saving Ratio (CSR)

Cost Saving Ratio is the measure of the percentage of the total cost of the queries saved due to hits in the Personalized Smart Cube. This measure is defined as:

$$CSR = \frac{\sum_i c_i h_i}{\sum_i c_i r_i} \tag{6.1}$$

where c_i is the cost of execution of query q_i without using the personalized smart cube, h_i is the number of times that the query was satified using the personalized smart cube and r_i is the total number of references to that query. This metric is also used in [18] for their experiments. Note that CSR is more appropriate metric than the common hit ratio: $\frac{\sum_i h_i}{\sum_i r_i}$ because query costs are known to vary widely. The drawback in the above definition for our case, is that it does not capture the different ways that a query q_i might "hit" the personalized cube. In the best scenario, q_i exactly matches a view V. In this case the savings is defined as c_i, where c_i is the cost of answering the query using the smart cube. In the case where a parent view or an ancestor view is used in answering q_i the actual savings depend on how "close" this materialized view is to the answer we want to produce. If c_f is the cost of querying the best such view v for answering q_i, the saving in this case is $c_i - c_f$. To capture all cases we define the savings provided by the personalized Smart Cube P for a query instance q_i as

$$s_i = \begin{cases} 0 & \text{if } q_i \text{ can not be answered by } P \\ c_i & \text{if there is an exact match for } q_i \text{ in } P \\ c_i - c_f & \text{If } f \text{ from } P \text{ was used to answer } q_i \end{cases} \tag{6.2}$$

using the above formula we define the Detailed Cost Saving Ratio as

$$DCSR = \frac{\sum_i s_i}{\sum_i c_i} \tag{6.3}$$

DCSR provides a more accurate measure than CSR for OLAP queries. CSR uses a "binary" definition of a hit, i.e. a query hits the personalized cube or not. For instance, if a query is computed using a cuboid from the Smart Cube with a cost $c_i = 200$ and from a materialized view in the personalized cube with cost $c_f = 120$, CSR will return a savings of 200 for the "hit", while DCSR will return the difference in savings which is 80 units based on the previous formula.

6.2 Experimental Results

In this section we present the results of our experiments. We study the effect of various parameter changes such as fragment size, dimensionality, row threshold and performance threshold (f) on the results of our algorithm. A comparative study is also done by comparing the result of our algorithm to some state-of-the-art approaches. The results are evaluated based on the criteria previously presented. The main aim of our result is to ascertain that our Personalized Smart Cube approach is able to answer the fundamental questions that were asked.

6.2.1 Storage Size

As noted earlier, the amount of memory is expressed as the number of rows of the materialized views set in order to simplify the computation of our data cubes. However, storage size here is the actual space used to store the materialized views. Using the synthetic database, we first analyzed the cost of storing the fragment cubes. As a notational convention, we use D to denote the number of dimensions, C the cardinality of each dimension, T the number of tuples in the database, F the size of the fragment, and S the skew or zipf of the data. First of all, to see the effect of dimensionality increase on the storage cost, we generated two synthetic dataset with the same number of rows but different cardinalities and varied the number of dimensions. For this experiment we used a row threshold (R) of 1000. Recall that the bigger the row threshold, the more views are prunned and therefore the fewer the number of materialized cuboids. The smaller the threshold, the larger the number of cuboids that are materialized and the larger the space required to store the cube. A row threshold value of 1,000 was selected by experimentation, since it took on average 2 seconds to execute a SELECT query on a thousand rows. This implies that storing views with rows less than 1,000 will not yield major query performance benefits.

Effect of Dimensionality on Storage Size

The charts in Figure 6.1, Figure 6.2, Figure 6.4 and Figure 6.4 depicts how dimensionality affect the space required to compute the data cube. In our first experiment, we were concerned with the cost of storing the materialized cube. Specifically, we evaluate how it scales as dimensionality grows. Figure 6.1 shows the effect as dimensionality increases from 20 to 80. The number of tuples in both datasets were 2,000,000. The first dataset, 50-C, has cardinality of 50, skew of 0, and fragment size of 8. The second dataset, 100-C, has cardinality of 100, skew of 2, and fragment size of 10. All values set by inspection. It will be noted that, in both instance, because of partitioning, the storage space grows linearly as dimensionality grows. This is expected, since additional dimensions only adds more fragments to the data cube, which are independent of other cubes. The difference in storage space between the two datasets had actually little to do with the cardinality and can be attributed to the fragment size; the smaller the fragment size, the fewer the number of materialized cubes are computed, although we do not compute full cube materialization for each fragment.

In Figure 6.2, we compared the Smart Cube with no Smart Views with that of Shell Fragment approach. Recall that the difference lies only in the computation of their local fragment cubes. While the Shell Fragment approach computes a full local cube for each

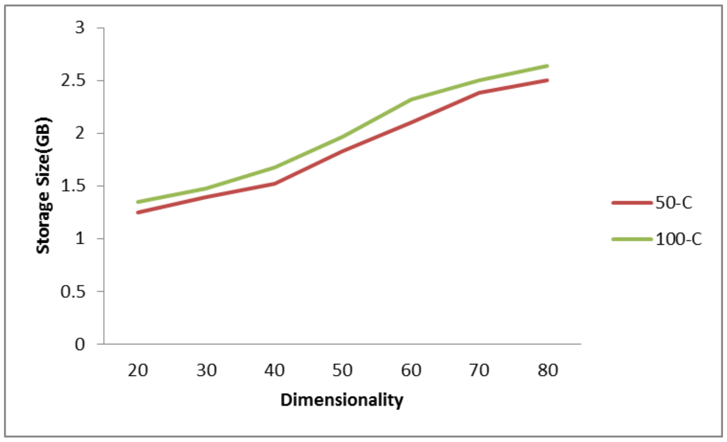


Figure 6.1: Storage size of materialized cube with no smart views: (50-C) T=2000000, C=50, S=0, F=8. (100-C) T=2000000, C=100, S=2, F=10

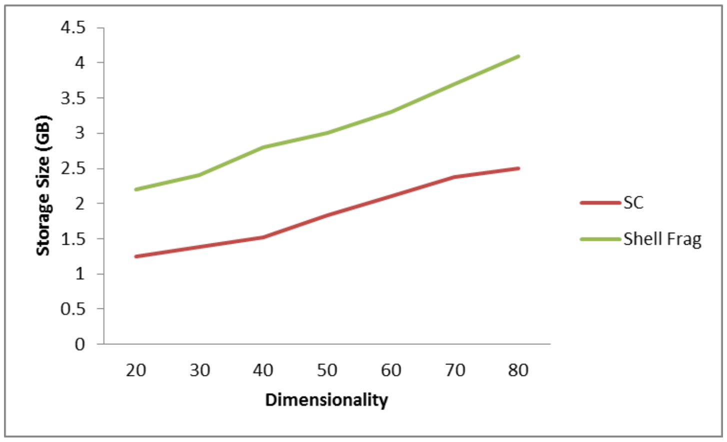


Figure 6.2: Storage size of smart cube with no smart view (SMNV) compared with Shell Fragment (SF): (SMNV) T=2,000,000, C=50, S=0, F=8. (SF) T=2,000,000 C=50, S=0, F=3

fragment, Smart Cube with no Smart Views computes a partially materialized cube for each fragment. The synthetic dataset that was used has 2,000,000 tuples, the cardinality is 50, the skew is 0. A fragment size of 8, selected by inspection, was used for the Smart Cube approach and fragment size of 3 was used for the Shell Fragment. The size of 3 was used for the Shell Fragment approach following [39], who showed that a fragment size of 3 produced the optimum solution. The size of the cube for the Shell Fragment was almost twice as big as the Smart Cube when we do not add the Smart Views. This was to be expected, since the Shell Fragment computed full cube materialization for each fragment. For example, for a table with 40 dimensions, using the Shell Fragment approach, 8 cuboids where computed for each fragment for a total of 106 cuboids in all.

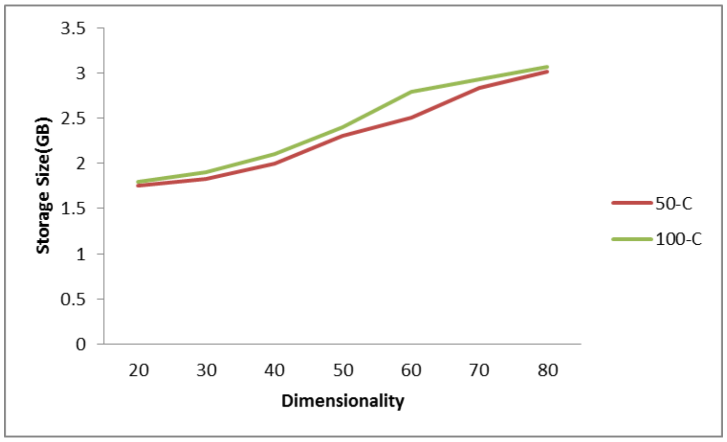


Figure 6.3: Storage size of materialized cube with smart views: (50-C) T=2000000, C=50, S=0, F=3. (100-C) T=2000000, C=100, S=2, F=10

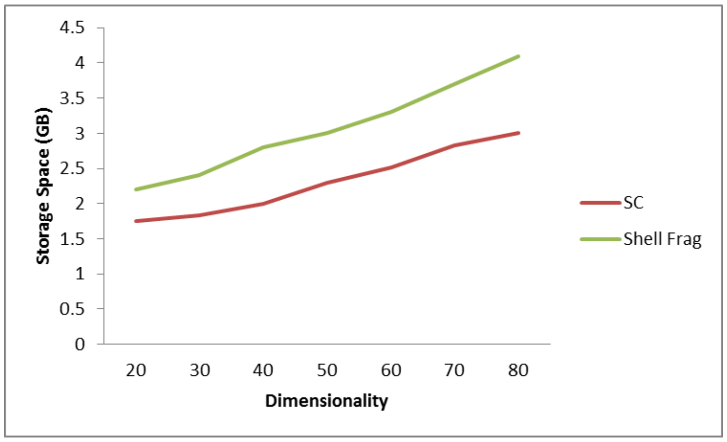


Figure 6.4: Storage size of Smart Cube with Smart View (SMNV) compared with Shell Fragment (SF): (SMNV) T=2,000,000, C=50, S=0, F=8. (SF) T=2,000,000 C=50, S=0, F=3

The Smart Cube approach, however, computed a total of 62 cuboid, which is 58.5% less than the Shell Fragment approach. A very small fragment size implies that the chances that a query will be computed from a single fragment is very small. That is, all high dimensional queries will always require online computation, which is very slow.

The two experiments in Figure 6.1 and Figure 6.2 show how dimensionality affects the Smart Cube storage size. We used the Smart Cube with no Smart Views in the two experiments, since our interest is to evaluate components of our approach systemically. As was noted in Chapter 4, Smart Views, although increasing the number of materialized views, also reduces the rate of online computation using Inverted Indexes. These are materialized views that a computed based on user query pattern and limited by amount of predefined storage allocated.

In Figure 6.4 we analyze the effect of dimensionality on the Smart Cube. In this experiment we again used the synthetic database as was used in the previous experiment. The first database 50-C, has cardinality of 50 and skew of 0 and the second database, 100-C, has cardinality of 100 and skew of 0. The number of tuples in each of the two databases is 2,000,000. The first database is partitioned into 8 fragments whereas the second database is partitioned in 10 fragments. These values were set by inspection. The result of this comparison shows a linear growth in storage size as dimensions grows even though materialized views from the Smart Views were added to the smart cube. As it can been seen from Figure 6.4, the size of the data cube computed for the 100-C database is slightly larger than that of the data cube for 50-C database. This is because of the fact that even though the fragment cubes for the 100-C database was larger, it had bigger fragments. Therefore, most of the auto-generated queries did not create new materialized views. Secondly the 100-C database has a cardinality of 100 as compared to the 50-C database which has a cardinality of 50. This implies that the views created by the 100-C database were larger and therefore fewer views were created for the allocated storage space. After adding the Smart View to the Smart Cube, we compared the Smart Cube to the Shell Fragment approach in terms of storage size as dimensionality grows. Figure 6.4 shows the result of the comparison. The database and parameter settings is the same as in the previous comparison. The addition of the Smart View to the Smart Cube is still smaller than the Shell Fragment cube. It follows that the size of the resulting Smart Cube is however, dependent on the storage space allocated for Smart Views.

Effect of Fragment Size on Storage Size

In [39] the authors showed that a fragment size between 2 and 4 strikes a good balance between storage space and computation time. However, the Shell Fragment approach was based on full materialization of each fragment cube. The increase in cube storage size grows exponentially as the fragment size becomes greater than 3. The question arises then, since we do not fully materialize our fragment cube, does changes in fragment size also changes the size of our storage?

The size of the smart cube grows linearly as fragment size increase. We observe from the experimental results that the number of materialized views for each fragment is usually less than the $\frac{2^n}{2}$, where the number of cuboids computed from a full cube materialization is 2^n. The difference between the two databases lies in the effect of dimensionality on the cube storage size, which from previous experiments we showed was also linearly correlated. The dimensional size of the databases were set by experimen-

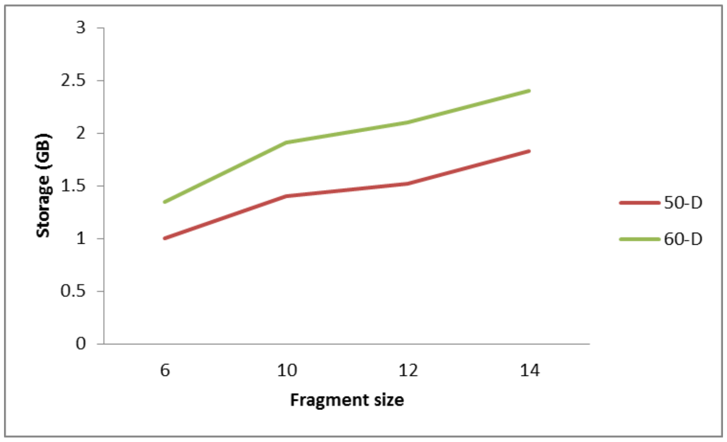


Figure 6.5: Storage size for Different Fragments: (50-D) T=2,000,000, D=50, C=50, S=0. (60-D) T=2,000,000, D=60, C=50, S=0

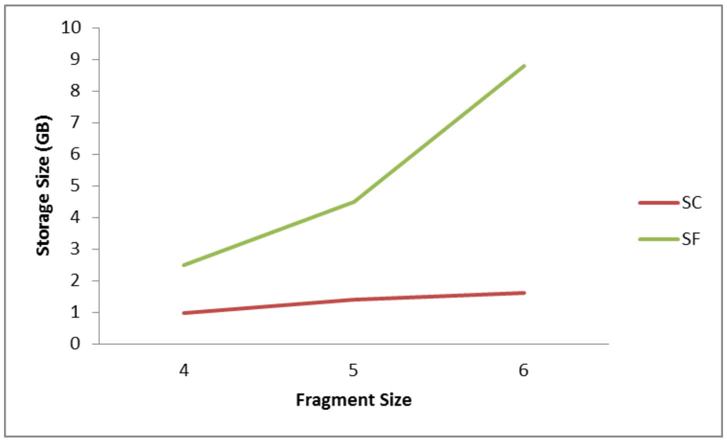


Figure 6.6: Storage size of SC compared with SF for Different Fragments: (50-D) T=2,000,000, D=50, C=50, S=0

tation to evaluate our algorithm on very large databases. The 60-D database from the Figure 6.5 requires larger storage than the data cube computed from the 50-D data set. In Figure 6.15 we compared our approach with the Shell Fragment approach. Using a synthetic dataset with 2,000,000 tuples, 50 dimensions, cardinality of 50 and skew of 0, we analyzed the effect of fragment size on the cube storage size. The Shell Fragment approach seemed to almost double the space required to store the fragment cube anytime the fragment size is increased by one.

Effect of Row Threshold on Storage Size

The row threshold prunes the search space by preventing further computation when the size of the parent node is less or equal to the row threshold. In Chapter 4, we saw that the larger the row threshold, the fewer the number of iterations or the shorter the computation time, and vise versa. This also implies that, the larger the row threshold, the fewer the number of materialized views. Now let us show experimentally how the row threshold affects the cube storage size.

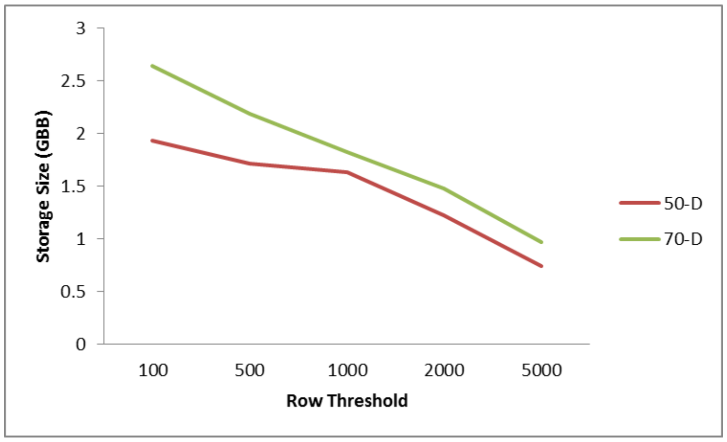


Figure 6.7: Storage size per row threshold: (50-D) T=2,000,000, D=50, C=50, S=0. (70-D) T=2,000,000, D=70, C=50, S=0

Figure 6.7 shows the relationship between the row threshold value and storage space. From Figure 6.7 it follows that there is an inverse relationship between the row threshold and the storage space used by the resultant data cube. Thus, the larger the row threshold the smaller the resultant data cube, and the smaller the row threshold the larger the resultant data cube. This is very obvious, since the higher the row threshold the fewer the number of materialized cuboids or views and hence the smaller the storage space required to store the views. The row threshold also has a direct effect on query latency. Determining the row threshold value is dependent on the domain in which the system is been used. For example, for certain systems, users can wait for about 15 to 30 seconds for a response to queries. Whereas, for certain mission critical systems, the users want response instantly and therefore cannot afford a 20 seconds delay. Therefore, choosing the right threshold is very critical to the response time of queries. It follows that there is no fixed value for the row threshold, since it is very much dependent on the DBMS and the specification of the server machine. As previously stated, another important criteria for evaluating a data cube is the performance in terms of query processing. Next we present our results in terms of query processing.

6.2.2 Performance Factor and Query Processing Time

Recall from Section 6.1.2 that, by performance factor, we are referring to the cost of evaluating a query posed to the system. In this section we will examine the query evaluation performance as a result of materialization. We will also analyze the implication of various factors on the speed of query processing. Our materialization algorithm reduces the number of materialized views using the row threshold, this implies that our algorithm generates fewer materialized views than other other approaches such as PickBorders and PBS algorithms. The question we address here is whether fewer materialized views imply a lower query execution performance.

Materialization and Performance Factor

We compared our approach to the PickBorders algorithm, with respect to their query evaluation performances. We chose PickBorders because our partial materialization algorithm of the local cube fragment extended the this algorithm. Recall that the performance factor is based on the cost of evaluating a query. The databases used in this experiment are the TPC-DS database and US Census database. In order to ensure a valid comparison is done, we first computed the fragment cube using PickBorders and then using our algorithm. The comparison was done using a single fragment cube. For this experiment, we executed both the Smart Cube and PickBorders algorithms using $f = 3.38$ and $f = 11.39$, where f is the performance threshold selected by inspection. We also selected a row threshold $R = 500$ by inspection for the Smart Cube algorithm.

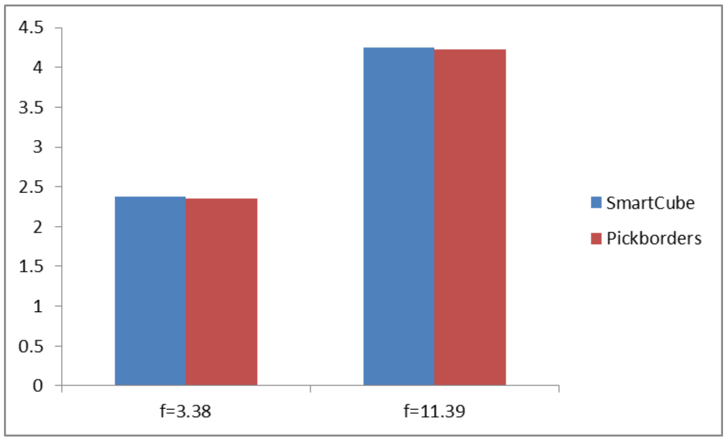


Figure 6.8: Performance Factor with $f = 3.38$ and $f = 11.39$

Figure 6.8 shows that the average performance of Smart Cube and the PickBorders algorithms are comparable. From the result it follows that, the lower the performance

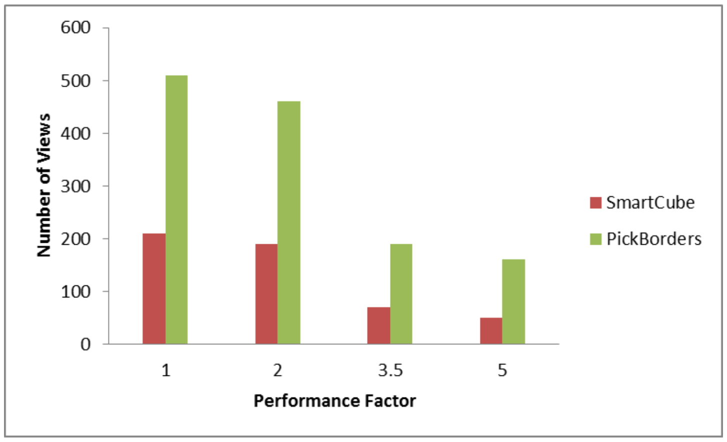


Figure 6.9: Performance Factor Distribution for TPC-DS dataset

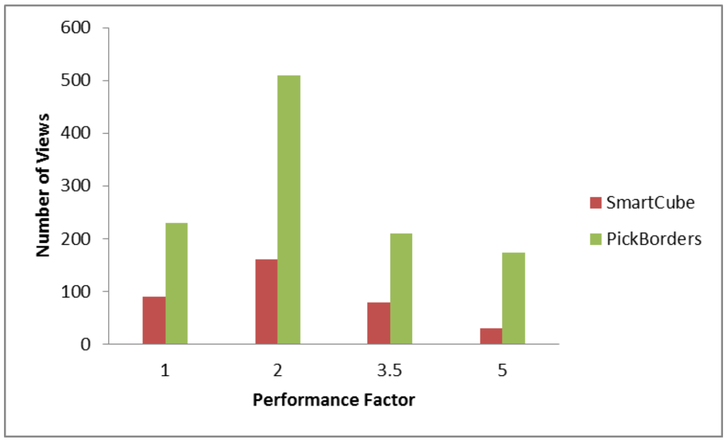


Figure 6.10: Performance Factor Distribution for US Census Data

threshold value f, the lower the cost of evaluating a query. This is because a lower f value mean more views will meet the threshold and hence will be materialized. The higher f value that is $f = 11.39$ implied that fewer views were materialized. This affected the average performance factor negatively. A more careful look at the distribution of the performance factors in Figure 6.9 and 6.10 shows that, even though the number of materialized cuboids in Smart Cube is less than that of the PickBorders, it does not have a negative impact on the performance. This further corroborate our assertion that a lower performance factor implied a higher number of materialized views. The first category represents the set of cuboids with performance factor 1; thus materialized views and views whose least materialized ancestor have the same size. The second category counts the cuboids with a performance factor in]1,2]. The third category counts the cuboids with a performance factor in]2, 3.5] and so on. Recall that the Smart Cube algorithm stores fewer number of cuboids than PickBorders algorithm, because it does not materialize those cuboids whose size is below the row threshold.

Query processing time

In order to assess how fast our algorithm is able to process queries, we executed different query profiles against our system and compared it with the state-of-the-art PickBorders and PBS algorithms. The database used was the TCP-DS database. The query profiles were used to evaluate the response time under different scenarios of query submissions. Using $f = 11.39$, selected by inspection, we conducted a test to compare the query processing speeds of the data cube generated. The query profiles selected include 10_90, 20_80, 50_80, 66_80 and $UNIFORM$. The $UNIFORM$ profile means that all query types have the same frequency. Smart Cube was configured using row threshold $\eta = 500$, and we used 30 distinct queries that were generated randomly. From Appendix A, a sequence of 400 queries was submitted, and we gathered the elapsed time to process all 400 queries according to each profile. The value of f and η were selected by inspection.

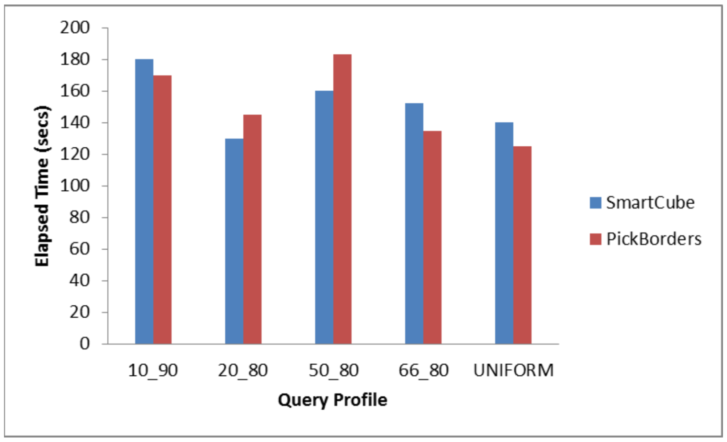


Figure 6.11: Runtime of Query Profile (SmartCube vs. PickBorders)

Figure 6.11 shows the results, which indicates that the runtime for executing queries using views created by Smart Cube is comparable to the runtime for executing queries using views selected by PickBorders, considering all query profiles. This further indicates that although SmartCube uses less materialized views and thus less storage space, PickBorders does not outperform Smart Cube in terms of overall query execution time. This, therefore, means that the selection of larger number of views does not necessarily imply a large time reduction to process queries.

We also compared the Smart Cube technique with the PBS algorithm. PBS is known to be a fast algorithm, and has broadly been used in comparative analysis, providing good results [29][34] [40]. In Figure 6.12, the result shows that executing queries using views

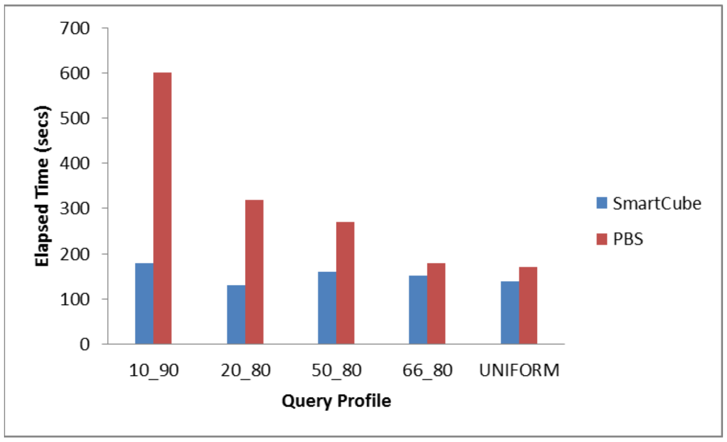


Figure 6.12: Runtime of Query Profile (SmartCube vs. PBS)

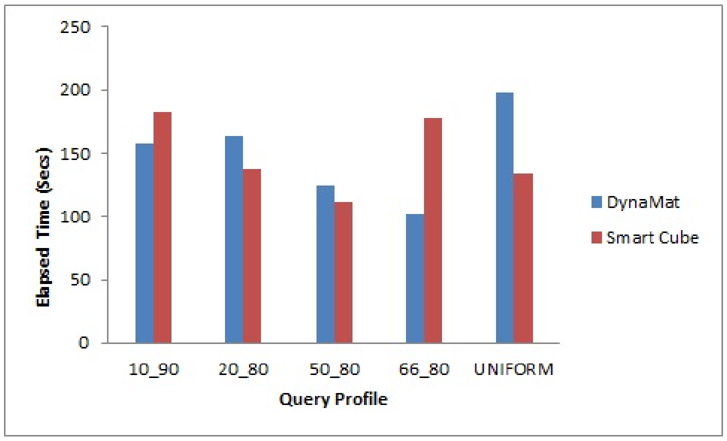


Figure 6.13: Runtime of Query Profile (SmartCube vs. DynaMat)

created using Smart Cube is more efficient than executing queries using views selected by the PBS algorithm. This is because PBS naively selects the smallest unselected view for materialization, without taking in to consideration the cost of materializing the view. The result further corroborated our initial assumption that the selection of large number of views does not necessarily imply a large time reduction to process queries.

In Figure 6.13, we compared our Smart Cube with DynaMat. The View Pool of DynaMat was simulated as was done in [17] [36]. That is, it utilizes dedicated secondary storage for managing the cache information (materialized views). The result showed that the two approaches produced comparable results in terms of query processing time. This can be attributed to the similarity between the DynaMat Pool and the Smart View approach. While the Smart View method materializes the most frequent view used to compute queries, the DynaMat algorithm materializes the results of the most frequent queries. DynaMat performs better in situations where the user queries are answered using the pool with the results of the exact same query already stored in the view fragment. The query processing time is similar when user queries are answered using the DynaMat Pool but the exact same query is not stored, and thus has to be computed from existing fragments. However, in general, more queries are answered using the Smart View since it stores the base view used for answering queries, while the DynaMat Pool stores query results. This implies that the chance that queries are answered using the Smart View are higher than when using the DynaMat Pool if the same storage size is allocated for both.

Fragment Size and query performance

In Figure 6.5 we saw the effect of fragment size on the storage size. Here, we will evaluate the effect of fragment size on query performance. In order to do so, we used the US Census database. In this section, we examine the implication of fragment size F on the speed of query processing.

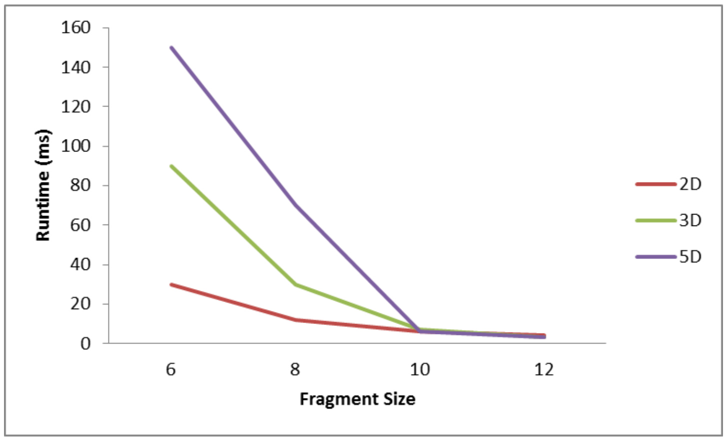


Figure 6.14: Average query time per 100 trials: (US Census data 1990) T=500K

Figure 6.14 shows the time needed to compute sub-cube queries or Range queries for different fragment sizes. The US Census 1990 database had 500,000 tuples, and 40 dimensions was used. The results shows fast query response time, with 150ms even when a 5D sub-cube is inquired; less for various type of queries. The results indicate that having a larger fragment size results in non trivial speedups, especially for high dimensional queries. This is because the probability of queries been answered from a single fragment increases with larger fragment size. Since the result in Figure 6.14 does not include the Smart View, it is essential that we minimize processing of the tid-lists using Inverted Indexes by ensuring all computation either happen in one or two

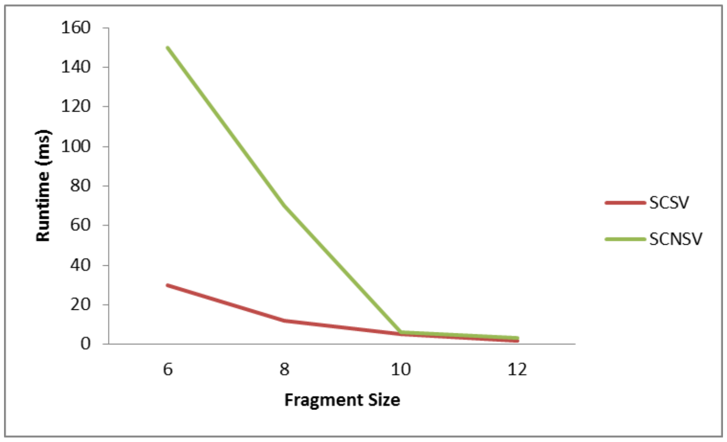


Figure 6.15: Average query time for Smart Cube with no Smart View (SCNSV) and Smart Cube with Smart View (SCSV): (US Census data 1990) T=500K

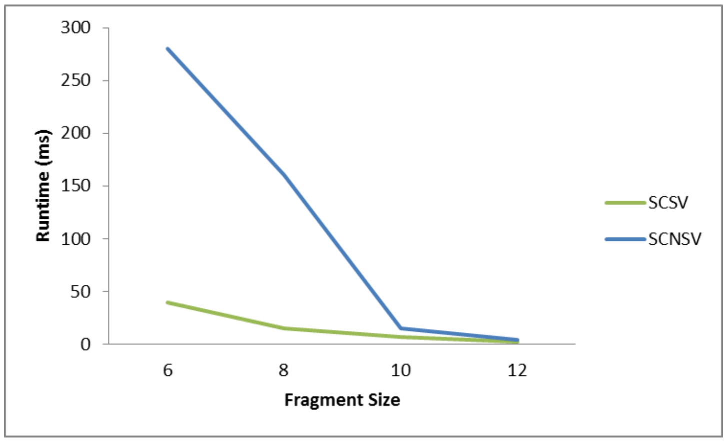


Figure 6.16: Average query time for Smart Cube with no Smart View (SCNSV) and Smart Cube with Smart View (SCSV): (TCP-DS data) T=3000,000

fragments. An important advantage to answering a query from a single fragment is that, for each fragment, multidimensional aggregates are already computed and stored.

As stated earlier, the data cube used for the experiment in Figure 6.14 does not include the Smart Views. However, Smart Views are expected to improve the performance of the Smart Cube by creating a *top* layer that links the individual fragments. In order to see the effect of the Smart View on query response time, we used the same US Census database as in Figure 6.14 and also the TCP-DS database. We computed our query on the Smart Cube with Smart View and compared the result with the Smart Cube with no Smart View. Figure 6.15 and Figure 6.16 shows the effect of different fragment size on our Smart Cube approach. As was expected, the Smart Cube with Smart View outperformed the Smart Cube with no Smart View in both databases. It follows that with the inclusion of the Smart Views, the performance of queries drastically improved; even in the worse case from Figure 6.15, the execution time is 30ms. This value improves as the fragment size increases. From our experiment, it follows that as fragment size increases, performance of the two approaches begins to converge until reaches a point where they converge. For the US Census database the query performance converged at a fragment size of 10, while for the TCP-DS database the query performance converged at fragment size greater than 12.

Personalization and query performance

As stated earlier, personalization provide the means by which a small subset of the data is updated dynamically for each user. This data subset ensures that, when users query the system, they get that small subset which is of interest to them, without superfluous data. However, because user queries are answered by the personalized views which are smaller in size, these queries are relatively faster than querying from the Smart Cube. We tested the personalization algorithm against the following query profiles: 10_90, 20_80, 50_80, 66_80 and $UNIFORM$. Some modifications were done to the queries to ensure we see the benefit of personalization on query performance. Thus, for the 10_90 profile we ensured that the 10% of the queries that repeated 90% of the time could be answered using the personalized cube. Also for the 20_80 profile, we ensured that the 20% of the queries that repeated 80% of the time could also be answered using the personalized views.

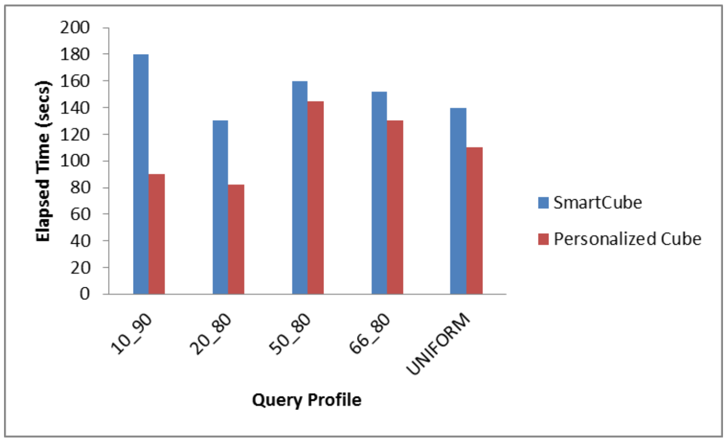


Figure 6.17: Runtime of Query Profile (SmartCube vs. Personal Cube)

The result of the experiment is illustrated in Figure 6.17. The graph shows an average of 43% improvement in the query performance when the Personalized Smart Cube is used over Smart Cube. The performance of queries in the 10_90 and 20_80 profiles, in particular, had about 50% improvement in query performance. This is because a high percentage of queries within these two profiles can be answered by the Personalized Smart Cube. The other query profiles also showed some moderate improvement in query performance. However, the improvements in query processing time were not that large because the queries within the profiles were distributed such that there is a fair percentage of queries that can be answered using the Personalized Smart Cube. Note that queries are either answered using the Personalized Cube or submitted to the Smart Cube for processing. This implies the total runtime for processing queries is dependent on how many queries within the query profile were executed by the Personalized Smart Cube and how many queries were submitted to the Smart Cube for processing. If the Personalized Smart Cube is able to answere a larger percentage of queries within the profile, then query processing time is reduced. However, if it is not, and the base Smart Cube answers a larger percentage of the queries, then query processing time increases.

6.2.3 Cost Saving Ratio (CSR)

Our Personalized Smart Cube algorithm is a dynamic algorithm that keeps monitoring user queries and materializing based on the performance gain. We compared our algorithm to the Virtual Cube algorithm using the TCP-DS database. We compared to the Virtual Cube because, to the best of our knowledge, it is the only other algorithm provides users with a relevant subset of the data based on their interest. To account for cost savings we executed the *uniform* query profile for different number of materialized views. The *uniform* query profile is made up of queries that target uniformly the Personalized Smart Cube and the base Smart Cube. This lack of locality of the queries represent the worst-case scenario for the Personalized Smart Cube, since its dynamic and needs to adapt on-the-fly to the incoming query pattern. For the Virtual Cube, this is not an issue, because it is static and therefore computes the virtual cubes in advance based on user interest I. The storage limit on both algorithms were set to the same value of 750MB.

The result in Figure 6.18 shows that generally cost savings increases as more views are materialized for each user. Initially when the number of materialized views are between 0 and 10, the Virtual Cube had a higher cost savings than the Personalized Smart Cube. This is because the Personalized Smart Cube was still learning the user query pattern and therefore computing most of the queries using the base Smart Cube. The Virtual Cube, on the other hand, precomputes the data cube using the user interest and a given cost model. Therefore the views selected for materialization is the optimum solution.

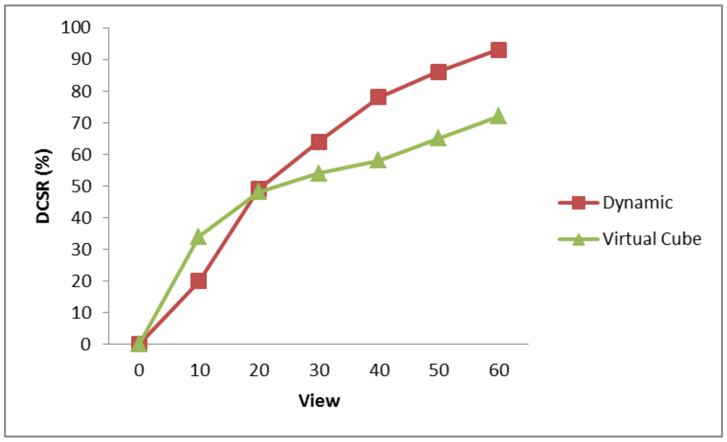


Figure 6.18: DCSR per view for uniform queries on views

When more views are materialized, the Personalized Smart Cube systematically studies the user query pattern and cost savings increases. This implies that the percentage number of queries answered using the Personalized Smart Cube increases compared to the Virtual Cube algorithm. Although the Virtual Cube also increases the cost savings with respect to increase in number of materialized views, this change is rather moderate, since all views are precomputed.

6.3 Discussion

We started by evaluating our algorithms based on cost and storage size. Dimensionality has always been a problem with data cube size, whether full or partial materialization is used. We therefore evaluated the effect of dimensionality on our Smart Cube. The result showed a linear increase in storage size as dimensionality increased, since increase in dimensionality only means addition of extra fragments. Next, we analyzed the effect of fragment size on the storage cost. For a full materialization of a fragment cube, increasing of fragment size above 4 implied an exponential increase in storage space. On the other hand, the results from our approach showed a linear increase in storage space as fragment size increases. This linear increase in storage space as dimensionality and fragment size increased was consistent, even when we added the Smart Views. Although dimensionality and fragment size are the main factors that influence the overall storage size of our Smart Cube, we further introduced the row threshold in our algorithm. The function of the row threshold is to determine the minimum size of views that should be stored. Since the value of the row threshold seems to influence the number of views that would be stored, we experimentally analyzed its effect on the overall storage size of our data cube. The result shows, as was expected, that the larger the threshold value, the fewer the number of cuboids and vise versa.

We further evaluated the performance factor and query evaluation time of the Personalized Smart Cube approach. Query processing time and the performance of a data cube is influenced by various factors, which include the materialization approach (i.e. the number of materialized views) and also the type of views that are materialized. The fragment size of each partition also influences the query evaluation time, since the larger the size the more the queries that can be answered from a single fragment. The performance of our approach was evaluated for different f values. We then compared the performance of our approach to that of the PickBorders algorithm. The result showed that, the performance of our approach is comparable to that of the PickBorders algorithm, but has the added advantage of materializing a fewer number of views and thus having a smaller size. This contradicts the belief that fewer materialized views imply slower query response time. This smaller number materialized views is as a result of the fact that our Smart Materialization algorithm reduces the number of views selected for materialization by eliminating views whose sizes are less than the row threshold. We also compared our approach to that of PickBorders and PBS algorithms, in terms of query processing time with different query profiles. The result showed that our algorithm outperformed the PBS algorithm which had higher number of materialized cuboids than our approach. We also compared our approach to the DynaMat algorithm and the results showed comparable performance. This is as a result of the fact that Smart View algorithm is an extension of DynaMat algorithm. The larger the fragment size the more storage required to store the data cubes and the better it performs with respect to query processing. The effect of fragment size on average query execution time was evaluated experimentally and the results showed that our technique generally performed well, even when Smart Views where not used. We also compared our system to the Shell Fragment approach. However, we did it for smaller fragment sizes, since computing larger fragment size using Shell Fragment approach was prohibitive.

Finally our proposed system include cube personalization, which provide users with the subset of data that is of most interest to them. This means that queries answered by personalization are supposed to be processed very fast, as compared to queries answered using the Smart Cube. We evaluated the query performance gain as a result of personalization. Another evaluation criteria used for our personalization algorithm was Cost Saving Ratio (CSR) or Detail Cost Saving Ratio (DCSR). CSR is the measure of the percentage of the total cost of the queries saved due to hits in the personalized data cube. CSR is a consequence of the assumption that personalized cubes provide fast response to queries than the Smart Cube. We therefore evaluated the query performance as a result of personalization. The result showed an average of 43% improvement in execution time. Finally, we compared our personalization approach to that of the Virtual Cube method. The result showed that, because of the dynamic nature of our personalization algorithm, the DCSR was higher than that of the Virtual Cube.

The strength of the Personalized Smart Cube computation method is the ability to first reduce the size of the data cube while guaranteeing performance. The second advantage is the provision of personalization in terms of user data requirements or users data interest. Personalization first gives users only that subset of data they are interested in and also reduces the query response time even further. The advantage of our methodology in reducing the size of the data cube can be attributed mainly to our smart materialization algorithm, which reduces the size of the data cube by terminating materialization at a threshold. This approach does not only reduce our Smart Cube size, but also reduces the time in computing the data cube.

6.4 Conclusion

In this chapter, we presented the results of evaluating the Personalized Smart Cube approach and various aspects of the cube construction phase. We studied the effect of running our algorithm on synthetic databases, real-world databases, and databases of different sizes and dimensionality. Our algorithm was evaluated on two major criteria, that is storage size and query performance. We compared our algorithms to the state-of-the-art methods and the results showed that our algorithm outperformed them based on the three main criteria for evaluation.

In the next chapter we summarize our thesis and present areas of future work.

Chapter 7

Conclusion

In this chapter we provide a summary of our work. We discuss the algorithms, results and contributions of our work. We finally suggest some future directions.

7.1 Discussion

Partitioning of databases before cube computation has been know to reduce the curse of dimensionality, by adding partition fragments as dimensionality increases. However, computing local fragment cubes using full cube materialization technique introduces additional problems. The main problem caused by such a technique is that, each fragment has an exponential storage and computational complexity since it computes all 2^d cuboids, where d is the number of dimensions. This means that, for very large fragment size, the computation time and storage space required for each fragment are prohibitive. Partial cube materialization, however, provides an optimal trade off between storage space and computation time complexity. Thus, with partial materialization, only the most beneficial cuboids are saved.

Several solutions have been proposed in order to find the most relevant subset of cuboids to store. The problem is then, with a constraint on the amount memory space that can be utilized, how to provide a subset of cuboids so that the cost is minimized. Most of the existing solutions suppose that the cuboids sizes are known in advance. This is often not the case. Other work propose an approximation algorithm whose performance guarantees that the gain of the returned solution cannot be less than a percentage of that of the optimal solution. In this method the notion of performance is obtained by comparing the returned solution to the ”worst” solution.

Data cubes are usually computed from data that are very large with high dimensionality. It provides a means to view the entire data from different perspectives. However, users are mostly interested in only a subset of the data. They want all OLAP cube operations to be done within the context of this subset. In order to meet such user needs, the data cube is personalized for the user. Personalization in OLAP cubes has been approached from different perspectives and context. Some perspective is related to *multidimensional data presentation* in which display data is adapted into constraint-based adaptable data cube. Other personalization approaches are based on associating user annotations to every piece of multidimensional data, while some are based on specifying relevant data according to user preferences.

This thesis provided a detailed cube computation technique that capitalized on partitioning, partial materialization and data cube personalization based on dynamic computation. In our materialization technique we addressed the problem of finding a set S that provides a cost no more than $minCost \times f$ given $f > 1$, where f is a threshold value provided by the user and *minCost* is the minimum cost of executing a query using a node that is not a parent. This algorithm reduces the number of materialized cuboids by ignoring views whose sizes are less then the *row threshold*. In other words, materializing such cuboids do not provide any extra performance benefit. In order to improve query processing time of all queries, including those computed from more than one fragment, we dynamically compute Smart Views by constantly monitoring incoming queries from users. The queries used for computing Smart Views are those that require more than one fragment to compute.

In order to evaluate the performance and quality of our proposed Personalized Smart Cube algorithm, we designed a prototype. We tested the system against the TPC-DS database, the US Census 1990 database, and a synthetic database. The experiments we conducted to evaluate the the Personalized Smart Cube shows, in general, that our proposed solution compared favorably with state-of-the-art methods. We analyzed the effect of data dimensionality and fragment size on the storage size and query evaluation time of the Smart Cubes. We compared the output of our algorithm with the Shell Fragment technique, evaluated on storage size. The results showed that our algorithm utilizes less storage than the Shell Fragment technique. We evaluated the performance of our algorithm in terms of query processing time by comparing it with the PickBorders and PBS algorithms. The results showed that, in addition to our algorithm materializes fewer views than both methods and thus computing a smaller cube, the query processing time was comparable with the PickBorders algorithm and outperformed the PBS algorithm. Finally, we evaluated our Personalized Cube on two criteria, namely query processing time and Detailed Cost Saving Ration (DCSR). We compared our Personalized Cube with the Virtual Cube algorithm. The results showed that initially, with a smaller number of materialized views the Virtual Cube slightly performed better in terms of

DCSR. However, as more views were added the Personalized Cube performed better than Virtual Cube due to the dynamic learning of user query pattern. The results also showed a drastic improvement in query processing time when Personalized Cubes are used, compared to Smart Cubes.

7.2 Thesis Contributions

Data cube computation, cube materialization techniques, and data cube personalization have been studied separately in the research community over the years. Our main contribution in this thesis is the combination of static and dynamic data cube computation techniques in order to reduce the space required to store the data cubes as well as the time required to process user queries. We also introduced a new algorithm that reduces the number of materialized cuboids by eliminating cuboids that do not add additional query performance, i.e. cuboids whose sizes are less than the row threshold.

The data cube computation technique involves partitioning the dimension space into disjoint sets called fragments and computing a data cube for each fragment. The computation of the local fragment cube is done using our Smart Materialization algorithm. For queries that require online computation using Inverted Indexes, we reduced the amount of online computation by using dynamic data cube computation technique. This technique dynamically monitors users incoming queries and materializes the query results based on some given criteria.

Our final contribution is that we proposed a dynamic data cube *personalization* algorithm that monitors incoming queries and based on user interest, materializes results based on a performance factor threshold. The benefit of cube personalization is that it provides users with the data subset that is of most interest to them by filtering out unnecessary information.

7.3 Future Work

The work presented in this thesis may be furthered in several other directions. Firstly, our personalization algorithm is only based on user query pattern. However, this can be extended to include situation awareness, where the context within which the query is issued will be taken into consideration. This may include special events and situations that might be of interest to the user. It will be interesting to explore the data for special drifts or changes that happen within the data, to be considered when personalizing the data cube.

Another possible direction is the following. Our algorithm serially go through each partition computing the data cube, even though the partitions are disjoint sets. We can therefore parallelize the computation of the fragment cube, so that cubes would be computed at the same time. This may considerably reduce the computation time of the Smart Cubes. Parallelization will not only reduce the computation time of the fragment cubes, but also the computation of the Smart Views will be drastically improved, since each fragment will be handled by a different process.

Another interesting area of future extension lies in the computation of the personalized cube. Our current approach uses the frequency of user access to a particular view to determine whether it should be materialized for the user, or not. However, we could investigate the use of newer and faster machine learning modules which would produce better results of identifying which cuboids would provide the most benefit when materialized.

Another suggestion for future work will be to go beyond personalized view materialization using relational databases and explore the possibility of computing the Personalized Smart Cube using in-memory databases. In-memory databases do not need materialization to improve performance. Therefore, views computed for each user will not require materialization, but will still provide fast, personalized answers.

Appendix A

List of Queries

Table A.1: One (1) Dimensional Queries using TCP-DS Database

MDX	User Levels	Filters
1. SELECT [Measures].[Sales Price] ON 0, [ITEM].[BRAND] ON 1 FROM [STORE SALES]	[ITEM].[BRAND]	No Filters
2. SELECT [Measures].[Sales Price] ON 0, [ITEM].[BRAND].[NEON] ON 1 FROM [STORE SALES]	[ITEM].[BRAND]	i_brand= "Neon"
3. SELECT [Measures].[Sales Price] ON 0, [ITEM].[CLASS].[DRESS] ON 1 FROM [STORE SALES]	[ITEM].[CLASS]	i_class= "dress"
4. SELECT [Measures].[Sales Price] ON 0, [ITEM].[CATEGORY].[MEN] ON 1 FROM [STORE SALES]	[ITEM].[CATEGORY]	i_category= "men"
5. SELECT [Measures].[Sales Price] ON 0, [STORE].[STORE NAME].[ABLE] ON 1 FROM [STORE SALES]	[STORE].[STORE NAME]	s_store_name= "able"
6. SELECT [Measures].[Sales Price] ON 0, [STORE].[MANAGER].[SCOTT SMITH] ON 1 FROM [STORE SALES]	[STORE].[MANAGER]	s_manager = "scott smith"

Continue on next page

Table A.1 – continued from previous page

MDX	User Levels	Filters
7. SELECT [Measures].[Sales Price] ON 0, [CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA] ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[CITY]	a_city= “Ottawa”
8. SELECT [Measures].[Sales Price] ON 0, [CUSTOMER].[CUSTOMER ADDRESS].[STATE].[CA] ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[STATE]	a_state= “CA”
9. SELECT [Measures].[Sales Price] ON 0, [DATE].[YEAR].[2005] ON 1 FROM [STORE SALES]	[DATE].[YEAR]	d_year= 2005
10. SELECT [Measures].[Sales Price] ON 0, [DATE.SEASON MONTH].[SEASON].[Winter].[January] ON 1 FROM [STORE SALES]	[DATE.SEASON MONTH].[MONTH]	d_season= “Winter” and d_month= “January”

Table A.2: Two (2) Dimensional Queries using TCP-DS Database

MDX	User Levels	Filters
1. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[BRAND]. Children} * {[STORE].[STORE NAME].[ABLE]. Children } ON 1 FROM [STORE SALES]	[ITEM].[BRAND], [STORE].[STORE NAME]	s_store_name= “able”
2. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[BRAND].[NEON]. Children} * {[STORE].[STORE NAME].[ABLE]. Children} ON 1 FROM [STORE SALES]	[ITEM].[BRAND], [STORE].[STORE NAME]	i_brand= “neon” and s_store_name= “able”
3. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[CLASS].[DRESS]. Children} * {[STORE].[MANAGER].[SCOTT SMITH]. Children} ON 1 FROM [STORE SALES]	[ITEM].[CLASS], [STORE].[MANAGER]	i_class= “dress” and s_manager= “ Scott Smith”

Continue on next page

Table A.2 – continued from previous page

MDX	User Levels	Filters
4. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[CATEGORY].[MEN]. Children} * {[STORE].[MANAGER].[JAMES CRETE]. Children} ON 1 FROM [STORE SALES]	[ITEM].[CATEGORY], [STORE].[MANAGER]	i_category= "men" and s_manager= "James Crete"
5. SELECT [Measures].[Sales Price] ON 0, {[STORE].[STORE NAME].[ABLE]. Children} * {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA] . Children} ON 1 FROM [STORE SALES]	[STORE].[STORE NAME], [CUSTOMER].[CUSTOMER ADDRESS].[CITY]	s_store_name= "able" and a_city= "Ottawa"
6. SELECT [Measures].[Sales Price] ON 0, {[STORE].[MANAGER].[SCOTT SMITH]. Children} * {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA] . Children} ON 1 FROM [STORE SALES]	[STORE].[MANAGER], [CUSTOMER].[CUSTOMER ADDRESS].[CITY]	s_manager = "scott smith" and a_city = "Ottawa"
7. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA]. Children} * {[ITEM].[CLASS].[DRESS]. Children} ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[CITY], [ITEM].[CLASS]	a_city= "Ottawa" and i_class= "dress"
8. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[STATE].[CA]. Children} * {[ITEM].[CLASS].[DRESS]. Children} ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[STATE], [ITEM].[CLASS]	a_state= "CA" and i_class = "dress"
9. SELECT [Measures].[Sales Price] ON 0, {[DATE].[YEAR].[2005]. Children}*{[ITEM].[CATEGORY].[MEN]. Children} ON 1 FROM [STORE SALES]	[DATE].[YEAR], [ITEM].[CATEGORY]	d_year= 2005 and i_category= "men"

Continue on next page

Table A.2 – continued from previous page

MDX	User Levels	Filters
10. SELECT [Measures].[Sales Price] ON 0, {[DATE.SEASON MONTH].[SEASON].[Winter].[January]. Children} * {[ITEM].[CATEGORY].[MEN]. Children} ON 1 FROM [STORE SALES]	[DATE.SEASON MONTH].[MONTH], [ITEM].[CATEGORY]	d_season= "Winter" and d_month= "January" and i_category= "men"

Table A.3: Three (3) Dimensional Queries using TCP-DS Database

MDX	User Levels	Filters
1. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY{{[ITEM].[BRAND] } * {[STORE].[STORE NAME].[ABLE], [STORE NAME].[OGT] } * {[DATE].[YEAR].[2005]}} ON 1 FROM [STORE SALES]	[ITEM].[BRAND], [STORE].[STORE NAME], [DATE].[YEAR]	s_store_name= "able" and d_year=2005
2. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[BRAND].[NEON]} * {[STORE].[STORE NAME].[ABLE]} ON 1 FROM [STORE SALES] WHERE [DATE].[YEAR].[2005]	[ITEM].[BRAND], [STORE].[STORE NAME], [DATE].[YEAR]	i_brand= "neon" and s_store_name= "able" and year=2005
3. SELECT [Measures].[Sales Price] ON 0, NON EMPTY{{[ITEM].[CLASS].[DRESS]} * {[STORE].[MANAGER].[SCOTT SMITH]} * {[DATE].[2005]:[DATE].[YEAR].[20010]}} ON 1 FROM [STORE SALES]	[ITEM].[CLASS], [STORE].[MANAGER], [DATE].[YEAR]	i_class= "dress" and s_manager= " Scott Smith" and d_year between 2005 and 2010

Continue on next page

Table A.3 – continued from previous page

MDX	User Levels	Filters
4. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY{Order(Order({[ITEM].[CATEGORY].[MEN]. Children}*{[STORE].[MANAGER].[JAMES CRETE]} * {[DATE].[YEAR].[2006], [DATE].[YEAR].[2007]}, [Measure].[Sales Price], BDESC),[DATE].CurrentMember.Name, BASC)} ON ROWS FROM [STORE SALES]	[ITEM].[CATEGORY], [STORE].[MANAGER], [DATE].[YEAR]	i_category= "men" and s_manager= "James Crete"
5. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY {{[STORE].[STORE NAME].[ABLE]} * {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA], [CUSTOMER].[CUSTOMER ADDRESS].[CITY].[KINGSTON]}} ON 1 FROM [STORE SALES] WHERE [ITEM].[CATEGORY].[MEN]	[STORE].[STORE NAME], [CUSTOMER].[CUSTOMER ADDRESS].[CITY], [ITEM].[CATEGORY]	s_store_name= "able" and a_city= "Ottawa" or "Kingston"
6. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY {{[STORE].[MANAGER].[SCOTT SMITH]. Children} * {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA] } * {[DEMOGRAPHICS].[GENDER][Male]}} ON ROWS FROM [STORE SALES]	[STORE].[MANAGER], [CUSTOMER].[CUSTOMER ADDRESS].[CITY], [DEMOGRAPHICS].[GENDER]	s_manager = "scott smith" and a_city = "Ottawa" and d_gender= "male"
7. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA]. Children} * {[ITEM].[CLASS].[DRESS]}*{ *{[DEMOGRAPHICS].[GENDER][Male]} ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[CITY], [ITEM].[CLASS], [DEMOGRAPHICS].[GENDER]	a_city= "Ottawa" and i_class= "dress" and d_gender= "male"

Continue on next page

Table A.3 – continued from previous page

MDX	User Levels	Filters
8. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[STATE].[CA]. Children} * {[ITEM].[CLASS].[DRESS]. Children} ON 1 FROM [STORE SALES] WHERE [DEMOGRAPHICS].[MARITAL STATUS].[Divorced]	[CUSTOMER].[CUSTOMER ADDRESS].[STATE], [ITEM].[CLASS], [DEMOGRAPHICS].[MARITAL STATUS]	a_state= "CA" and i_class = "dress" and d_marital_status= "divorced"
9. SELECT [Measures].[Sales Price] ON COLUMNS, {[DATE].[YEAR].[2005]. Children} * {[ITEM].[CATEGORY].[MEN]. Children} ON ROWS FROM [STORE SALES] WHERE [DEMOGRAPHICS].[MARITAL STATUS].[Divorced]	[DATE].[YEAR], [ITEM].[CATEGORY], [DEMOGRAPHICS].[MARITAL STATUS]	d_year= 2005 and i_category= "men" and d_marital_status= "divorced"
10. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY{ {[DATE.SEASON MONTH].[SEASON].[Winter].[January]} * {[ITEM].[CATEGORY].[MEN]} * { [CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA]}} ON ROWS FROM [STORE SALES]	[DATE.SEASON MONTH].[MONTH], [ITEM].[CATEGORY], [CUSTOMER].[CUSTOMER ADDRESS].[CITY]	d_season= "Winter" and d_month= "January" and i_category= "men" and a_city= "Ottawa"

Table A.4: Four (4) Dimensional Queries using TCP-DS Database

MDX	User Levels	Filters
1. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY{{[ITEM].[BRAND] } * {[STORE].[STORE NAME].[ABLE], [STORE NAME].[OGT] } * {[DATE].[YEAR].[2005]} * { [DEMOGRAPHICS].[GENDER].[Male]}} ON 1 FROM [STORE SALES]	[ITEM].[BRAND], [STORE].[STORE NAME], [DATE].[YEAR], [DEMOGRAPHICS].[GENDER]	s_store_name= "able" and d_year=2005 and d_gender= "male"

Continue on next page

Table A.4 – continued from previous page

MDX	User Levels	Filters
2. SELECT [Measures].[Sales Price] ON 0, {[ITEM].[BRAND].[NEON]} * {[STORE].[STORE NAME].[ABLE]} * { [DEMOGRAPHICS].[GENDER].[Male]} ON 1 FROM [STORE SALES] WHERE [DATE].[YEAR].[2005]	[ITEM].[BRAND], [STORE].[STORE NAME], [DATE].[YEAR], [DEMOGRAPHICS].[GENDER]	i_brand= “neon” and s_store_name= “able” and year=2005 and d_gender= “male”
3. SELECT [Measures].[Sales Price] ON 0, NON EMPTY{{[ITEM].[CLASS].[DRESS]} * {[STORE].[MANAGER].[SCOTT SMITH]} * {[DATE].[YEAR].[2005]:[DATE].[YEAR].[20010]} * { [DEMOGRAPHICS].[GENDER].[Female]}} ON 1 FROM [STORE SALES]	[ITEM].[CLASS], [STORE].[MANAGER], [DATE].[YEAR], [DEMOGRAPHICS].[GENDER]	_class= “dress” and s_manager= “ Scott Smith” and d_year between 2005 and 2010 and d_gender= “female”
4. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY{Order(Order({[ITEM].[CATEGORY].[MEN]. Children}*{[STORE].[MANAGER].[JAMES CRETE]} * {[DATE].[YEAR].[2006], [DATE].[YEAR].[2007]} * { [DEMOGRAPHICS].[GENDER].[Female]},[Measure].[Sales Price], BDESC),[DATE].CurrentMember.Name, BASC)} ON ROWS FROM [STORE SALES]	[ITEM].[CATEGORY], [STORE].[MANAGER], [DATE].[YEAR], [DEMOGRAPHICS].[GENDER]	i_category= “men” and s_manager= “James Crete” and d_gender= “femal”

Continue on next page

Table A.4 – continued from previous page

MDX	User Levels	Filters
5. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY {{[STORE].[STORE NAME].[ABLE]} * {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA], [CUSTOMER].[CUSTOMER ADDRESS].[CITY].[KINGSTON]} * { [DEMOGRAPHICS].[GENDER].[Female]}} ON 1 FROM [STORE SALES] WHERE [ITEM].[CATEGORY].[MEN]	[STORE].[STORE NAME], [CUSTOMER].[CUSTOMER ADDRESS].[CITY], [DEMOGRAPHICS].[GENDER], [ITEM][CATEGORY]	s_store_name= “able” and a_city= “Ottawa” or “Kingston” and d_gender= “femal”
6. SELECT NON EMPTY {[Measures].[Sales Price]} ON COLUMNS, NON EMPTY {{[STORE].[MANAGER].[SCOTT SMITH]. Children}*{[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA] } * {[DEMOGRAPHICS].[GENDER].[Male]} * {[PROMO].[PROMO NAME].[CALLY]}} ON ROWS FROM [STORE SALES]	[STORE].[MANAGER], [CUSTOMER].[CUSTOMER ADDRESS].[CITY], [DEMOGRAPHICS].[GENDER], [PROMO].[PROMO NAME]	s_manager = “scott smith” and a_city = “Ottawa” and d_gender= “male” and p_promo_name= “cally”
7. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[CITY].[OTTAWA]. Children} * {[ITEM].[CLASS].[DRESS]}*{ *{[DEMOGRAPHICS].[GENDER].[Male]} * {[PROMO].[PROMO NAME].[CALLY]} ON 1 FROM [STORE SALES]	[CUSTOMER].[CUSTOMER ADDRESS].[CITY], [ITEM].[CLASS], [DEMOGRAPHICS].[GENDER], [PROMO].[PROMO NAME]	a_city= “Ottawa” and i_class= “dress” and d_gender= “male” and p_promo_name= “cally”
8. SELECT [Measures].[Sales Price] ON 0, {[CUSTOMER].[CUSTOMER ADDRESS].[STATE].[CA]. Children} * {[ITEM].[CLASS].[DRESS]} * { [DATE].[YEAR].[2005]:[DATE].[YEAR].[20010]} ON 1 FROM [STORE SALES] WHERE [DEMOGRAPHICS].[MARITAL STATUS].[Divorced]	[CUSTOMER].[CUSTOMER ADDRESS].[STATE], [ITEM].[CLASS], [DEMOGRAPHICS].[MARITAL STATUS], [DATE].[YEAR]	a_state= “CA” and i_class = “dress” and d_marital_status= “divorced” and d_year between 2005 and 2010

Table A.5: Sample Queries using US Census 1990 Database

MDX	**User Levels**	**Filters**
1. SELECT NON EMPTY {[Measures].[Income 1]} ON COLUMNS, NON EMPTY{{[CENSUS].[SEX].[MALE] } * {[CENSUS].[MARITAL].[MARRIED] }} ON 1 FROM [US CENSUS]	[CENSUS].[SEX], [CENSUS].[MARITAL]	Sex= “male” and marital= “married”
2. SELECT [Measures].[Income 1] ON 0, {[CENSUS].[DISABLED].[YES]} * {[CENSUS].[CITIZEN].[YES]} ON 1 FROM [US CENSUS]	[CENSUS].[DISABLED], [CENSUS].[CITIZEN]	Disabled = “yes” and citizen = “yes
3. SELECT [Measures].[Income 1] ON 0, NON EMPTY{{[CENSUS].[DISABLED].[NO]} * {[CENSUS].[SEX].[FELMAL]}} ON 1 FROM [US CENSUS]	[CENSUS].[DISABLED], [CENSUS].[SEX]	Disabled = “no” and sex = “female”
4. SELECT NON EMPTY {[Measures].[Income 1]} ON COLUMNS, NON EMPTY {{[CENSUS].[MARITAL].[SINGLE]}*{[CENSUS].[CITIZEN].[YES] }} ON 1 FROM [US CENSUS]	[CENSUS].[MARITAL] [CENSUS].[CITIZEN]	Marital= “single” and Citizen= “yes”
5. SELECT NON EMPTY {[Measures].[Income 1]} ON COLUMNS, NON EMPTY {{[CENSUS].[CITIZEN].[NO]}} ON ROWS FROM [US CENSUS]	[CENSUS].[CITIZEN]	Citizen = “no”

Bibliography

[1] Mondrain Schema Workbench. `http://mondrian.pentaho.com/documentation/workbench.php`,, 2013.

[2] Sameet Agarwal, Rakesh Agrawal, Prasad M Deshpande, Ashish Gupta, Jeffrey F Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *Proceeding of Very Large Database Conference*, volume 96, pages 506–521, 1996.

[3] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[4] Kamel Aouiche and Daniel Lemire. A comparison of five probabilistic view-size estimation techniques in OLAP. In *Proceedings of the ACM Tenth International workshop on Data Warehousing and OLAP*, DOLAP '07, pages 17–24, New York, NY, USA, 2007. ACM.

[5] Thilini Ariyachandra and Hugh Watson. Key organizational factors in data warehouse architecture selection. *Decis. Support Syst.*, 49(2):200–212, May 2010.

[6] K. Bache and M. Lichman. UCI Machine Learning Repository. `http://archive.ics.uci.edu/ml`, 2013.

[7] Ricardo A. Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[8] Elena Baralis, Stefano Paraboschi, and Ernest Teniente. Materialized views selection in a multidimensional database. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 156–165, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[9] Daniel Barbará and Mark Sullivan. Quasi-cubes: exploiting approximations in multidimensional databases. *ACM SIGMOD Record*, 26(3):12–17, 1997.

[10] Ladjel Bellatreche and Kamel Boukhalfa. Yet another algorithms for selecting bitmap join indexes. In *Proceedings of the 12th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK'10, pages 105–116, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, pages 359–370, New York, NY, USA, 1999. ACM.

[12] Truls A. Bjrklund, Nils Grimsmo, Johannes Gehrke, and system Torbjrnsen. Inverted indexes vs. bitmap indexes in decision support systems. CIKM '09, pages 1509–1512, New York, NY, USA, 2009. ACM.

[13] Jose A. Blakeley, Per-Ake Larson, and Frank Wm Tompa. Efficiently updating materialized views. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, SIGMOD '86, pages 61–71, New York, NY, USA, 1986. ACM.

[14] S. Chaudhuri, U. Dayal, and V. Ganti. Database technology for decision support systems. *Computer*, 34(12):48–55, 2001.

[15] Surajit Chaudhuri and Umeshwar Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, March 1997.

[16] Andresson da Silva Firmino, Rodrigo Costa Mateus, Valéria Cesário Times, Lucidio Formiga Cabral, Thiago Luís Lopes Siqueira, Ricardo Rodrigues Ciferri, and Cristina Dutra de Aguiar Ciferri. A novel method for selecting and materializing views based on OLAP signatures and grasp. *JIDM*, 2(3):479–494, 2011.

[17] F. Dehne, M. Lawrence, and A. Rau&#45;Chaplin. Cooperative caching for grid&#45;enabled olap. *Int. J. Grid Util. Comput.*, 1(2):169–181, December 2009.

[18] Prasad Deshpande, Karthikeyan Ramasamy, Amit Shukla, and Jeffrey F. Naughton. Caching multidimensional queries using chunks. In *In Proceedings of the ACM SIGMOD Conference on Management of Data*, pages 259–270, 1998.

[19] Steven Geffner, Divyakant Agrawal, and Amr El Abbadi. The dynamic data cubes. In *Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '00, pages 237–253, London, UK, UK, 2000. Springer-Verlag.

[20] Matteo Golfarelli, Dario Maio, and Stefano Rizzi. Applying vertical fragmentation techniques in logical design of multidimensional databases. In *Data Warehousing and Knowledge Discovery*, volume 1874 of *Lecture Notes in Computer Science*, pages 11–23. Springer Berlin Heidelberg, 2000.

[21] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data Cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Min. Knowl. Discov.*, 1(1):29–53, January 1997.

[22] Himanshu Gupta. Selection of views to materialize in a data warehouse. In *Proceedings of the 6th International Conference on Database Theory*, ICDT '97, pages 98–112, London, UK, UK, 1997. Springer-Verlag.

[23] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proceedings of the Thirteenth International Conference on Data Engineering*, ICDE '97, pages 208–219, Washington, DC, USA, 1997. IEEE Computer Society.

[24] Himanshu Gupta and Inderpal Singh Mumick. Selection of views to materialize under a maintenance cost constraint. In *Proceedings of the 7th International Conference on Database Theory*, ICDT '99, pages 453–470, London, UK, UK, 1999. Springer-Verlag.

[25] Peter J. Haas, Jeffrey F. Naughton, S. Seshadri, and Lynne Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 311–322, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[26] A. Hajmoosaei, M. Kashfi, and P. Kailasam. Comparison plan for data warehouse system architectures. In *Data Mining and Intelligent Information Technology Applications (ICMiA), 2011 3rd International Conference on*, pages 290–293, 2011.

[27] Jiawei Han, Micheline Kamber, and Jian Pei. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2011.

[28] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *ACM SIGMOD Record*, volume 30, pages 1–12. ACM, 2001.

[29] Nicolas Hanusse, Sofian Maabout, and Radu Tofan. A view selection algorithm with performance guarantee. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '09, pages 946–957, New York, NY, USA, 2009. ACM.

[30] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, SIGMOD '96, pages 205–216, New York, NY, USA, 1996. ACM.

[31] M. Ibrahim, J. Küng, and N. Revell. *Database and Expert Systems Applications: 11th International Conference, DEXA 2000 London, UK, September 4-8, 2000 Proceedings*. Database and Expert Systems Applications: 11th International Conference, DEXA 2000, London, UK, September 4-8, 2000 : Proceedings. Springer, 2000.

[32] W. H. Inmon, Claudia Imhoff, and Greg Battas. *Building the Operational Data Store*. John Wiley & Sons, Inc., New York, NY, USA, 1995.

[33] Antoaneta Ivanova and Boris Rachev. Multidimensional models: constructing data cube. In *Proceedings of the 5th international conference on Computer systems and technologies*, CompSysTech '04, pages 1–7, New York, NY, USA, 2004. ACM.

[34] Panos Kalnis, Nikos Mamoulis, and Dimitris Papadias. View selection using randomized search. *Data Knowl. Eng.*, 42(1):89–111, July 2002.

[35] Ralph Kimball and Margy Ross. *The Data Warehouse Toolkit: The Complete Guide to Dimensional Modeling*. John Wiley & Sons, Inc., New York, NY, USA, 2nd edition, 2002.

[36] Yannis Kotidis and Nick Roussopoulos. Dynamat: a dynamic view management system for data warehouses. *SIGMOD Rec.*, 28(2):371–382, June 1999.

[37] Laks VS Lakshmanan, Jian Pei, and Jiawei Han. Quotient cube: How to summarize the semantics of a data cube. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 778–789. VLDB Endowment, 2002.

[38] Jianzhong Li, Doron Rotem, and Jaideep Srivastava. Aggregation algorithms for very large compressed data warehouses. In *Proceeding of the 25th VLDB Conference*, pages 651–662. Morgan Kaufmann Publishers, 1999.

[39] Xiaolei Li, Jiawei Han, and Hector Gonzalez. High-dimensional OLAP: a minimal cubing approach. In *Proceedings of the Thirtieth International Conference on Very large databases - Volume 30*, VLDB '04, pages 528–539. VLDB Endowment, 2004.

[40] Zhou Lijuan, Ge Xuebin, Wang Linshuang, and Shi Qian. Research on materialized view selection algorithm in data warehouse. In *Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on*, volume 2, pages 326–329, 2009.

[41] Yannis Manolopoulos, Alexandros Nanopoulos, Apostolos N. Papadopoulos, and Y. Theodoridis. *R-Trees: Theory and Applications (Advanced Information and Knowledge Processing)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[42] Konstantinos Morfonios, Stratis Konakas, Yannis Ioannidis, and Nikolaos Kotsis. Rolap implementations of the data cube. *ACM Comput. Surv.*, 39(4), November 2007.

[43] Derek Munneke, Kirsten Wahlstrom, and Mukesh Mohania. Fragmentation of multidimensional databases. In *Proceedings of 10th Australasian Database Conf*, pages 153–164, 1999.

[44] Thomas P. Nadeau and Toby J. Teorey. Achieving scalability in OLAP materialized view selection. In *Proceedings of the 5th ACM International Workshop on Data Warehousing and OLAP*, DOLAP '02, pages 28–34, New York, NY, USA, 2002. ACM.

[45] Patrick O'Neil, Elizabeth O'Neil, Xuedong Chen, and Stephen Revilak. Performance evaluation and benchmarking. pages 237–252. Springer-Verlag, Berlin, Heidelberg, 2009.

[46] Patrick O'Neil and Dallan Quass. Improved query performance with variant indexes. *SIGMOD Rec.*, 26(2):38–49, June 1997.

[47] Patrick E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, London, UK, UK, 1989. Springer-Verlag.

[48] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems.* McGraw-Hill, Inc., New York, NY, USA, 3 edition, 2003.

[49] Franck Ravat and Olivier Teste. Personalization and OLAP databases. In Stanislaw Kozielski and Robert Wrembel, editors, *New Trends in Data Warehousing and Data Analysis*, volume 3 of *Annals of Information Systems*, pages 1–22. Springer US, 2009.

[50] Mirek Riedewald, Divyakant Agrawal, Amr El Abbadi, and Renato Pajarola. Space-efficient data cubes for dynamic environments. In *Proceedings of DaWaK Conference*, volume 1874 of *Lecture Notes in Computer Science*, pages 24–33. Springer, 2000.

[51] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 116–125, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[52] Jayavel Shanmugasundaram, Usama Fayyad, and Paul S Bradley. Compressed data cubes for OLAP aggregate query approximation on continuous dimensions. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 223–232. ACM, 1999.

[53] Amit Shukla, Prasad Deshpande, and Jeffrey F. Naughton. Materialized view selection for multidimensional datasets. In *Proceedings of the 24th International Conference on Very Large Data Bases*, VLDB '98, pages 488–499, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[54] Amit Shukla, Prasad Deshpande, Jeffrey F. Naughton, and Karthikeyan Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *Proceedings of the 22th International Conference on Very Large Data Bases*, VLDB '96, pages 522–531, San Francisco, CA, USA, 1996. Morgan Kaufmann Publishers Inc.

[55] Yannis Sismanis, Antonios Deligiannakis, Nick Roussopoulos, and Yannis Kotidis. Dwarf: Shrinking the petacube. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, SIGMOD '02, pages 464–475, New York, NY, USA, 2002. ACM.

[56] Yannis Sismanis and Nick Roussopoulos. The dwarf data cube eliminates the highy dimensionality curse. Technical Report UMIACS-TR-2003-120, University of Maryland, December 2003.

[57] Zohreh Asgharzadeh Talebi, Rada Chirkova, Yahya Fathi, and Matthias Stallmann. Exact and inexact methods for selecting views and indexes for OLAP performance improvement. In *Proceedings of the 11th International Conference on Extending Database Technology: Advances in Database Technology*, EDBT '08, pages 311–322, New York, NY, USA, 2008. ACM.

[58] TPC. Transaction processing performance council (1.1.0), April 2013. `http://www.tpc.org/tpcds/`.

[59] Panos Vassiliadis and Timos Sellis. A survey of logical models for OLAP databases. *SIGMOD Rec.*, 28(4):64–69, December 1999.

[60] Ganesh Viswanathan and Markus Schneider. OLAP formulations for supporting complex spatial objects in data warehouses. In *Proceedings of the 13th International Conference on Data Warehousing and Knowledge Discovery*, DaWaK'11, pages 39–50, Berlin, Heidelberg, 2011. Springer-Verlag.

[61] Wei Wang, Jianlin Feng, Hongjun Lu, and J.X. Yu. Condensed cube: an effective approach to reducing data cube size. In *Data Engineering, 2002. Proceedings. 18th International Conference on*, pages 155–165, 2002.

[62] Robert Wrembel and Christian Koncilia. *Data Warehouses And OLAP: Concepts, Architectures And Solutions.* IRM Press, 2006.

[63] Kesheng Wu, Ekow J. Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM Trans. Database Syst.*, 31(1):1–38, March 2006.

[64] Hao Yan, Shuai Ding, and Torsten Suel. Inverted index compression and query processing with optimized document ordering. In *Proceedings of the 18th International Conference on World Wide Web*, WWW '09, pages 401–410, New York, NY, USA, 2009. ACM.

[65] Dehui Zhang, Shaohua Tan, Dongqing Yang, Shiwei Tang, Xiuli Ma, and Lizheng Jiang. Dynamic construction of user defined virtual cubes. In *Proceedings of the 6th International Conference on Next Generation Information Technologies and Systems*, NGITS'06, pages 287–299, Berlin, Heidelberg, 2006. Springer-Verlag.

[66] Yihong Zhao, Prasad M. Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 159–170, New York, NY, USA, 1997. ACM.

[67] Justin Zobel and Alistair Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 38(2), July 2006.

[68] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar RAM-CPU cache compression. In *Proceedings of the 22nd International Conference on Data Engineering*, ICDE '06, pages 59–59, 2006.