High Performance Analytics with the R^3-cache

Ruhan Sayeed

A Thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fullfilment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

April, 2009

Your file Votre référence
ISBN: 978-0-494-63335-9
Our file Notre référence
ISBN: 978-0-494-63335-9

ABSTRACT

High Performance Analytics with the R^3-cache

Ruhan Sayeed

Contemporary data warehouses now represent some of the world's largest databases. As these systems grow in size and complexity, however, it becomes increasingly difficult for brute force query processing approaches to meet the performance demands of end users. Certainly, improved indexing and more selective view materialization are helpful in this regard. Nevertheless, with warehouses moving into the multi-terabyte range, it is clear that the minimization of external memory accesses must be a primary performance objective. In this thesis, we describe the R^3-cache, a natively multi-dimensional caching framework designed specifically to support sophisticated warehouse/OLAP environments. R^3-cache is based upon an in-memory version of the R-tree that has been extended to support buffer pages rather than disk blocks. A key strength of the R^3-cache is that it is able to utilize multi-dimensional fragments of previous query results so as to significantly minimize the frequency and scale of disk accesses. Experimental results demonstrate significant performance improvements relative to simpler alternatives.

ACKNOWLEDGEMENTS

First, let me express my gratitude to my supervisor Todd Eavis for his support and encouragement. His experience and knowledge in the subject matter has helped me enormously in pursuing this research.

Many thanks also to all my friends for all the fun and relief they brought during the most testing period of my life. Thanks for all the help and suggestion they gave me regarding the thesis and defense preparation.

Finally, I will be forever indebted to my parents and my brother for the enormous sacrifice they made to make sure that I can pursue this thesis.

To my brother without whom this thesis wouldn't have been possible

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Multi-dimensional caching

OLAP (Online Analytical Processing) is becoming increasingly important in the context of today's data driven world. In fact, complex data analysis is crucial for any long-term business plan and OLAP systems and applications provide us with efficient techniques for managing and accessing the associated data stores. As with any database system, caching has the potential to play a huge role in this context. In particular, an efficient caching system can dramatically enhance the overall performance of an OLAP DBMS. In data warehouses (the backing data stores for OLAP servers), queries often take a long time to execute due to their complex structure and the huge volume of data involved. Moreover, because of the nature of the data warehouse environment, the same sets are queried repeatedly. This may occur when a single user submits a batch of related queries or even when different users happen to submit the same kinds of queries. Consequently, query response time can be greatly improved by caching either full or partial results.

As is the case with DBMS systems in general, memory-resident caches can and should be used to improve query response. Unlike traditional DBMS caches, however, the fundamental structure of OLAP queries can be exploited to dramatically improve the capabilities of the cache manager. Specifically, OLAP queries tend to be "cubic" in nature; in other words, the most common pattern is a multi-dimensional range query that defines a contiguous hyper-cubic region in the data space. That being the case, existing buffer pages in OLAP-aware caches can subsume future queries that fall within the hyper-cubic boundary. More powerfully still, new queries that spatially overlap multiple existing pages can be dynamically transformed so as to minimize the number of disk accesses required. Efficiently exploiting this notion of the geometric cache is a crucial performance concern for real world OLAP servers.

A number of proposals have been put forward thus far for managing OLAP query caches. But very few of those solutions are capable of handling OLAP/data warehouse queries in higher dimensions (e.g., 3–7). More importantly still, when caching methods have been proposed, they are typically associated with highly unrealistic, over simplified test environments.

1.2 A new caching model

Previous work on OLAP caching has focused on either *table level* caching or *query level* caching. Table level caching is suitable only for static schemes, while query level caching can be applied to more dynamic environments. One of the major hurdles for query level caching, however, is that it is only effective when a new query is subsumed *completely* by a previous query result.

As a result of these shortcomings, Multi-dimensional OLAP (MOLAP) caching was introduced. MOLAP, instead of using relational tuples to organize the data, creates a multidimensional cube (e.g, an array) where attribute values serve as indexes in the cube space. MOLAP will be discussed in detail in the next chapter. At this point, however, we note that due to the native data structure, a MOLAP cube has the potential to grow very large in higher dimensions, especially when large cardinalities are encountered. So, though it works well in lower dimensions, in the absence of a very good compression routine, it is likely to perform poorly in higher dimensions.

In this thesis, we propose instead a solution based on Relational OLAP (ROLAP). Specifically, we have developed the R^3-cache, a mechanism based on the **non-overlapping R-tree** (NOR Tree) model. Whereas R-trees generally have overlapping child objects, a NOR tree consists of non-overlapping *leaf node* objects. This allows us to limit the scale of the tree traversal and, consequently, to use an efficient algorithm to search and find data-points. Our R^3-cache, a *cube*-oriented caching framework, supports the spatial manipulation of both full and partial query matches. The functionality of the R^3-cache is of course based upon the traditional R-tree, a multi-dimensional disk-based indexing structure often seen in research and industrial settings. The caching model is, in turn, fully integrated into the Sidera OLAP DBMS, a fully parallelized "shared nothing" server that seeks to provide robust, high performance analytics for today's massive decision support environments.

We note that although functionality like insertion, deletion, searching, and node-splitting in the R^3-cache is similar to that of the R-tree, the two models differ in some vital ways because of the non-overlapping structure of the leaf-nodes. As a simple example, consider the node-splitting mechanism. In a R-tree data structure, the leaf nodes can overlap one another, while in a R^3-cache we must use an extended algorithm to split the overlapping nodes in order to make them non-overlapping. This, in turns, allows our searching mechanism to be more efficient.

In addition, we use efficient cache management techniques to replace old — and relatively ineffective — data with new data. Cache management is a vital part of our R^3-cache solution. Because the hypercubic cache results can be both large and complex in nature, a poorly maintained cache can degenerate into a slow, inefficient server component if pro-active cache management is not undertaken. In this respect, because cache size is finite, it is important to replace old, unused cache data with more frequently used data. In fact, our cache management has two integral parts: **cache replacement** and **cache update**. We use meta-data such as query size, frequency, and query response time to calculate the *viability* of the nodes and to find the *hot spots* in the cache. Based on these calculations, we perform the required replacement and update operations. As a result, we believe our R^3-cache design, in combination with our cache management policy, makes an effective and efficient OLAP caching model.

1.3 Experimental results

We have done extensive testing using data sets ranging in size from 100,000 to 10 million records, skew rates of 50 to 90 percent, and dimension counts in the 2–7 range. Most of the experimental results are time-based, but some are also based on the cache hit-ratio.

As a concrete example, we conducted experiments by running 600 queries in the data warehouse DB, first with the R^3-cache, then without the cache. Our results show query cost in the non-cached system to be three to five times more expensive relative to the R^3-cache model. For gauging the effectiveness of our cache management policy, we ran additional tests on the R^3-cache, with and without the cache update process.

In the absence of an effective cache management policy, our results show no significant performance improvement as the cache size and query number increases. However, with the addition of our cache management policy, query time is essentially cut in half.

We have also performed a comparative analysis by first implementing an alternative MOLAP solution [10]. We used the same platform and data sets for both systems and again executed 600 queries against the data warehouse DB, first with the MOLAP cache, then with our own R^3-cache. Though the performance in two dimensions is nearly the same, our Relational OLAP system performed significantly better in higher dimensions, often by a factor of three or more.

1.4 Thesis structure

The rest of the thesis is organized as follows. Chapter 2 presents an overview of the OLAP architecture and its operation. It also provides a brief overview of Relational OLAP, Multi-dimensional OLAP and the R-Tree. Chapter 3 contains the main contribution of the thesis. It discusses the structure of the NOR-tree, as well as the details of the larger caching framework. We also discuss the issues faced during the implementation process and how we addressed them. Chapter 4 presents the results of our experimental analysis. As noted, we include a comparative study of our system relative to a MOLAP alternative. Finally, Chapter 5 provides a brief summary of the contributions of the thesis and points to possible future work.

Chapter 2

Background Material

2.1 Introduction

Data warehousing is becoming an increasingly important focus of the database industry. One of the primary reasons behind this increased attention is OLAP, a processing model that is capable of supporting sophisticated Decision Support Systems (DSS). Conventional database technologies like Online Transaction Processing (OLTP), which are used for more direct database access, generally lack these capabilities. However, the requirements associated with Decision Support lead to additional constraints and demands for OLAP tools. We will discuss some of these issues in this chapter.

We will begin with a general overview of OLAP technologies. Here, we will describe multi-dimensional data models typical of OLAP, with a particular emphasis on Relational OLAP (ROLAP) and Multi-dimensional OLAP (MOLAP). We describe their underlying data structures, as well as their strengths and weaknesses. Finally, we look at the previous works carried out in the area of data warehouse caching and discuss their strengths and weaknesses.

2.2 Decision support system

Historically, as corporations grew in scale and sophistication, so too did the size of their stored electronic data. Data generation and analysis became more complex and inefficient with the standard DBMS systems found in OLTP environments. Eventually, database designers realized that decision support systems should be constructed as separate entities, independent of the front line OLTP systems. This allowed DSS applications to provide efficient, flexible analysis of raw historical data gathered from a variety of different sources and formats (e.g., RDBMS, text file, XML).

The DSS model can be sub-divided into three broad categories [11]. We discuss these models below.

- **Information Processing** facilitates basic query and reporting functions. Only the simplest forms of analysis are performed. In comparison to OLAP, IP systems are quite limited in scope and complexity.

- **OLAP** is an extension of Information Processing in terms of its capabilities. That being said, it's quite different in its design. With IP, data is retrieved from *live* databases, whereas OLAP uses data from many sources that is subsequently consolidated into a single multi-dimensional data store. OLAP uses a data warehouse as the underlying storage system, and then typically augments the raw data with (i) an associated OLAP server and (ii) graphical OLAP tools on the client computing systems through which users can analyze data from selected, hierarchical dimensions [35]. We will discuss OLAP in greater detail in Section 2.4.

- **Data Mining** is the last logical step in the data analysis process. In short, data mining — at least in the data warehousing context — refers to the search for patterns or trends in data across various dimensions. In contrast to user-driven OLAP analysis, the data mining process is driven by the data itself.

2.3 Formal definition of the data warehouse

Before we delve deeper into the OLAP model, it's important to first define the data warehouse (DW) and it's architecture since the data warehouse is an integral part of virtually any OLAP system. According to Bill Inmon [19], the most accurate and useful definition for the term is that "Data warehousing is a process, not a product, for assembling and managing data from various sources for the purpose of gaining a single, detailed view of part or all of a business." As previously noted, a data warehouse is a separate physical entity consisting of historically consolidated data gathered from various sources. OLAP uses the data from the data warehouse to provide analysis across broad time frames.

In fact, in the same paper, Inmon gave a more formal definition of the data warehouse, one that includes the following properties.

- **Subject oriented.** Data is organized so that all the data elements relating to the same real-world event or object are linked together.
- **Time variant.** Changes in the data are tracked and recorded so that reports can be produced, showing changes over time.
- **Non Volatile.** Data is never over-written or deleted, but retained for future reporting.

- **Integrated.** The data warehouse contains data from most or all operational applications of an organization, and this data is made consistent.

2.3.1 Data warehouse architecture

A data warehouse can be seen as a four-tier architecture [7]. Figure 2.1 provides a simple illustration. At the very bottom of the "DW stack" we find all possible sources of data such as operational databases, text files, XML, etc. Data is transmitted from this level to the next level using a process known as ETL (Extract, Transform and Load). Data from various sources is extracted, transformed, harmonized and then loaded using sophisticated ETL applications [22]. The next level contains the data warehouse proper, as well as its associated data marts (i.e., sub-divisions). This is the point at which data in the DBMS is structured into a multi-dimensional architecture known as a *star schema* or *snow-flake schema*. At this stage, data is amenable to analysis and reporting. At the third level, we see the OLAP engine which further prepares or transforms the raw data contained in the underlying warehouse. Core OLAP functionality including slicing, dicing, roll up, drill down, and pivot would be supported here. Finally, at the highest level, we find various end user tools that managers and decision makers can use to perform complex decision making queries.

2.3.2 Schema

There are two primary database schemes that are commonly used for logical representation of the OLAP tables.

- **Star Schema.** Figure 2.2 shows the logical representation of the Star Schema. In a Star Schema, there are two kinds of tables — Fact tables and Dimension

Figure 2.1: The data warehouse architecture

tables. Typically there is one fact table in a given Star Schema, with the remainder being dimension tables. A fact table consists of all the primary keys from each of the dimension tables, referred to as *feature* attributes, along with one of more *measure* attributes. A measure attribute is a "process of interest" such as Total Sales or Average Revenue. By contrast, Dimension tables consist of a series of descriptive attributes that define some characteristic of the Dimension (e.g., Customer Name, Phone Number). In practice, fact tables tend to be relatively "narrow" but extremely large, with many fields redundantly recorded throughout the table. This process of allowing redundant data is called *de-normalization.* While normalization is ordinarily performed in OLTP systems, so as to eliminate redundancy, OLAP systems explicitly encourage de-normalized tables as it improves performance by minimizing expensive joins.

Figure 2.2: Star Schema

- **Snowflake Schema.** A Snowflake Schema is similar to a Star Schema but it does allow normalization for some dimension tables. In those cases, the fact table joins with the normalized table which, in turn, joins with the additional tables. For example, in Figure 2.3, the Product dimension is normalized so as to create a new table called Category.

2.4 OLAP

Large corporations have, over the years, accumulated vast amounts of data in the form of databases, text files and various other external resources. This data can be processed and analyzed to gather knowledge and find historical trends. OLAP differs from OLTP systems in this regard, because whereas OLTP systems are generally used for day-to-day detail-level transactions, OLAP systems are used to analyze data and trends across a much broader time frame. In addition, OLTP systems tend to deal

Figure 2.3: Snow-flake Schema

with a large number of small queries, while OLAP systems typically support a small number of very complex queries. This of course implies that the requirements and functionality of an OLAP system are much different than those of an OLTP system.

2.4.1 Formal definition of OLAP

The term "OLAP" was first used by E.F. Codd in 1992. In a paper entitled "Providing OLAP (on-line analytical processing) to user-analysts: An IT mandate" [19], Codd proposed 12 basic features that should be found in any OLAP application.

1. For a system to qualify as a relational database management system (RDBMS), that system must use its relational facilities (exclusively) to manage the database.

2. All information in the database is to be represented in one and only one way, namely by values in column positions within rows of tables.

3. All data must be accessible with no ambiguity. This rule is essentially a restatement of the fundamental requirement for primary keys. It says that every individual scalar value in the database must be logically addressable by specifying the name of the containing table, the name of the containing column and the primary key value of the containing row.

4. The DBMS must allow each field to remain null (or empty). Specifically, it must support a representation of missing information and inapplicable information that is systematic, distinct from all regular values (for example, distinct from zero or any other number, in the case of numeric values), and independent of data type. It is also implied that such representations must be manipulated by the DBMS in a systematic way.

5. The system must support an online, inline, relational catalog that is accessible to authorized users by means of their regular query language. That is, users must be able to access the database's structure (catalog) using the same query language that they use to access the database's data.

6. The system must support at least one relational language that (a) has a linear syntax (b) can be used both interactively and within application programs (c) supports data definition operations (including view definitions), data manipulation operations (update as well as retrieval), security and integrity constraints, and transaction management operations (begin, commit, and rollback).

7. All views that are theoretically updatable must be updatable by the system.

8. The system must support set-at-a-time insert, update, and delete operators.

This means that data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table.

9. Changes to the physical level (how the data is stored, whether in arrays or linked lists etc.) must not require a change to an application based on the structure.

10. Changes to the logical level (tables, columns, rows, and so on) must not require a change to an application based on the structure. Logical data independence is more difficult to achieve than physical data independence.

11. Integrity constraints must be specified separately from application programs and stored in the catalog. It must be possible to change such constraints as and when appropriate without unnecessarily affecting existing applications.

12. The distribution of portions of the database to various locations should be invisible to users of the database. Existing applications should continue to operate successfully: (a) when a distributed version of the DBMS is first introduced and (b) when existing distributed data are redistributed around the system.

13. If the system provides a low-level (record-at-a-time) interface, then that interface cannot be used to subvert the system, for example, bypassing a relational security or integrity constraint.

2.4.2 OLAP functionality

Below we list the most basic capabilities of an OLAP system, what we sometimes refer to as **Five Function OLAP**.

- **Pivot:** By doing a Pivot, a user can view the multi-dimensional data structure from different axes. In Figure 2.4, we see an example of a pivot operation on a simple three-dimensional OLAP cube.
- **Roll-up:** Roll-up helps the user to get a coarser summary along a particular dimension *hierarchy*. In Figure 2.5, we can see an example of a roll-up (and drill-down) operation on an OLAP cube. Figure A shows the full cube, with a roll-up operation on Figure A producing Figure B.
- **Drill-down:** A Drill-down gives the user a more detailed summary along a dimension hierarchy. In Figure 2.5, a drill-down operation on Figure B produces Figure C.
- **Slice:** Slicing allows a user to extract a segment of the original cube along a particular dimension. This helps to narrow down the focus of interest. In Figure 2.6, Figure A represents the full cube. After a user runs a Slice query, we are left with the sliced cube in Figure B.
- **Dice:** A Dice allows a user to slice the data set along more than one dimension. In Figure 2.7, Figure A again represents the full cube. The diced cube is shown in Figure B.

2.4.3 Physical model

In terms of physical OLAP models, there are two basic forms — ROLAP (Relational OLAP) and MOLAP (Multi-dimensional OLAP). The decision as to which to use often depends on the objectives of the user and the scale or capacity of the data warehouse.

Figure 2.4: The Pivot function

Figure 2.5: Roll-up and Drill-down functions

Figure 2.6: The Slice function

Figure 2.7: The Dice function

Figure 2.8: A two-dimensional MOLAP cube

- **MOLAP (Multi-dimensional OLAP).** MOLAP is the more widely used model in commercial settings. Here, data is basically stored in a multi-dimensional array. As shown in Figure 2.8, the length of an array in a given dimension is equivalent to the cardinality of the OLAP dimension associated with that axis. So, for example, if an attribute Date is represented by the x-axis of the array, then the size of the x-axis will be 30 because the cardinality of the Date dimension is 30 (in this example). Now, if we associate the attribute Product with the y-axis and we subsequently need to query the Sales table for the product "food" purchased on date "5", all we need to do is to look up the result at index position [5, 2].

  Advantages of MOLAP

  - Query performance is very fast due to its implicit multidimensional indexing.
  - Storage is quite efficient for low dimension data sets.

  Disadvantages of MOLAP

  - Data loading and conversion from relational DWs can be expensive for large input sets.
  - Storage becomes prohibitively expensive with higher dimension counts and large cardinalities since empty cells are stored by default

- **ROLAP (Relational OLAP).** As the name suggests, ROLAP organizes the data in standard relational structures. Since traditional RDBMS systems use the same structure for tables and relations, ROLAP can easily be used with the standard Star Schema.

  Advantages of ROLAP

  - In comparison with MOLAP, ROLAP tends to scale more gracefully since it only includes records that are actually in the database.
  - It loads much faster than MOLAP since RDBMS systems uses the same relational mappings.
  - Any SQL tool can access the ROLAP data warehouse directly, which makes it a lot more administrator-friendly than MOLAP

  Disadvantages of ROLAP

  - Because SQL is used for querying, performance may be limited by the constraints of the SQL language itself.

Figure 2.9: 4D-Lattice

- Performance is less impressive, particularly in lower dimensions.
- Explicit multi-dimensional indexing is required.

2.4.4 Data cube

Data warehouse users are generally mid-level or high-level managers who are most comfortable working in an intuitive, graphical environment. In this context, data is represented as a multidimensional *data cube* whose 2-D, 3-D, or even higher-dimensional sub cubes can be explored in an interactive fashion. As previously noted, cube attributes can be of two types — features and measures. In Figure 2.8, the Date and Location fields are feature attributes, while Sales Price is the measure.

If a data warehouse has n dimensions, the number of possible sub-cubes or *cuboids* is exactly 2^n. In short, these 2^n cuboids represent all possible combinations of feature attribute in the cube space. Put another way, each of these summary views provides a different perspective on the measure value being studied. Often, we represent the full set of cuboids in what is known as the Cube Lattice. The important point to note about the lattice is that a given child view in the lattice can always be computed from a parent view since the parent essentially represents a slightly more detailed aggregation of the same data. In the lattice depicted in Figure 2.9, the relationship between these cuboids can be seen quite clearly (each letter represents a dimension). Here, we have a four-dimensional space. At the top of the lattice, the *base cuboid* ABCD contains the most detailed information. At the bottom is the cuboid with the most aggregation (i.e., just a single total value).

2.5 Caching for DSS environments

The interest in OLAP as a research pursuit grew out of the seminal data cube paper by Gray et al [15]. Subsequently, researchers focused on fundamental construction algorithms for OLAP cubes, with a particular emphasis on the efficient generation of all 2^d cuboids in the d-dimensional space [5, 2].

2.5.1 Static versus Dynamic Caching

More recently, data warehouse caching has been a subject of some interest among academics. A number of the previous solutions can be classified in two general categories — *Static pre-computed caching* and *dynamic pre-computed caching.*

- **Static pre-computed caching** uses an upfront algorithm to pre-compute aggregate data in anticipation of future user queries. In this case, the decision as to which cube is to be pre-computed does not depend on the eventual user queries themselves.

In [13], an algorithm is presented that is used to create an initial query execution plan. It then selects the most frequently used nodes for pre-computation. An aggregation lattice is also employed in order to identify the best cost saving ratio in terms of additional node storage. Because the node count in this model is bounded as $O(n^3)$, heuristics are used to reduce the size of the lattice.

- **Dynamic pre-computed caching** tries to adapt to user queries by dynamically optimizing the pre-computed aggregates. In [32], the authors present a dynamic caching scheme based on relational OLAP. They describe a cache manager model that can dynamically decide which results need to be cached and which results need to be removed from the existing cache. The manager can search the cache for either an exact-match query or queries that subsume the entire searching query.

  When a new query is executed, the cache manager at first looks for an exact-match set in the existing cache. If the exact-match is not found, a "Split" algorithm is run on the query to produce a transformed query. Subsequently, the existing cache is searched for an exact match or for previous query results that may subsume the transformed query. If more than one such entry is found in the cache, then the one with the minimum query response time is chosen. A *query attachment graph* is maintained to store the relationship between the transformed query and the selected result set.

  Figure 2.10 provides a graphical snapshot of the cache manager. The *Split Algorithm* divides the Query Q into two queries $Q1$ and $Q2$. In this case, $Q1$ is

Figure 2.10: Basic architecture

Figure 2.11: BASE Query and SLICE query

Figure 2.12: The Lattice

the base cuboid for the query Q, while $Q2$ is essentially just the original Q. For example, as shown in Figure 2.11, $Q1$ would be the BASE Query and $Q2$ would be the SLICE query. If the BASE query $Q1$ has already been materialized in the cache, then the query $Q2$ can be answered directly from the cache.

But if the BASE query has not yet been materialized, then the query must be answered from the least costly ancestor cuboid. Using the lattice defined in Figure 2.12, if a query is made on the Course-Semester-Student cube but the base query for this cube has not yet been materialized, then any of the parent views for the cube can be used to calculate the query result.

Since caches are stored in memory, their size is quite limited relative to the disk resident data warehouse. To deal with this fact, the authors of the Split Algorithm introduce a comprehensive cache replacement and admission algorithm to maintain the cache so that only the most queried regions are stored in the cache. They do this using a complex formula that exploits the number of hits and a least recently used (LRU) cache policy . A "profit" value is associated with each of the stored cache objects which are then stored in the query attachment graph. Depending on the profit value, queries are either removed or inserted into the cache system.

The Split Algorithm approach improves on a number of previously proposed ROLAP based caching solutions [36, 20, 31, 8, 1, 9]. For example, [31] supports subsumption query caching without support for aggregate level caching. This severely restricts the effectiveness of the technique. [8, 1, 9] support some aggregation during caching but their supporting algorithms are very costly in practice.

That being said, the biggest weakness of the Split Algorithm is the absence of partial caching. The mechanism only supports exact query matching and subsumption query matching, but it has no mechanism to search for partial results. Partial query searching is especially useful in higher dimensions. Moreover, as the dimension count grows, the probability of subsumption by an existing query drops as well. Finally, the efficiency and effectiveness of the system depends on the BASE cuboid being fully materialized in the cache. This might be okay for low dimensions and small spaces. However, it can be extremely costly in more realistic OLAP environments.

2.5.2 Physical models

In addition to the static versus dynamic distinction, we can also classify caching solutions with respect to their physical structure. Here, we can consider table based, query based and chunk based (MOLAP) options. We briefly discuss the first two, then turn to a more detailed discussion of chunk based caching, a model that we use for comparative evaluation.

- **Query level** partitioning schemes cache the data in a relational data format(ROLAP). Very few OLAP caching solutions have been proposed using query level partitioning. The few that have been proposed have been quite limited. For example, in [32] and [33], query results must either be an exact match or must be completely subsumed by the existing cache. It was argued in [10] that solutions lacking the partial match feature eventually end up using the same data in many different queries which, of course, results in significant storage waste.

- **Table based** scheme [18, 16] cannot be partitioned to generate additional cubes and therefore they are classified as table-based partitioning. All the summary tables are defined without restriction so it's not possible to perform OLAP functionalities like Slice and Dice over the summary tables. This restriction makes this method quite inflexible.

Chunk based caches

Chunk based partitioning schemes cache the data in some form of multi-dimensional array. As opposed to table based schemes, a chunk based model caches smaller and smaller cubes within its physical store. Deshpande and Naughton, for example, implemented data warehouse caching using chunks [10]. To date, this is the most comprehensive implementation of its type.

At a high level, a chunk based cache can be described as follows:

- Chunks are carved out of a multi-dimensional array. As previously noted, the cardinalities of each dimension in a multi-dimensional data structure form an inclusive range along each dimension. A subset of values along each dimension forms a *chunk*. In effect, a chunk is just a sub-cube extracted from the larger space. Figure 2.14 illustrates a chunk defined in three dimensions — Course, Student and Semester.

- Query results are stored in chunks when a new query is issued. Subsequent queries are then divided into two sets. One set consists of the queries that need to be retrieved from the data warehouse and the other set represents queries that are already in the cache. Figure 2.13 illustrates this division. Here queries Q1 and Q3 are retrieved from the database, while Q2 is retrieved from the Cache itself.

Not surprisingly, the idea for chunk based caching comes from the more general MOLAP (multi-dimensional OLAP) model, where multi-dimensional data is stored in an array format. In contrast to the relational or query based schemes, where data is stored in a *row major* format, the chunking systems divide data into sub-cubes and store them in an array. Formally, we can define the size of a chunk as $C = \prod_{i=1}^{n} D_i$, where n is the number of dimensions and $D_i \in D$, with D being the set of ranges for all the dimensions. Figure 2.14 illustrates the basic chunk storage structure.

Chunks being moved to/from the data warehouse need be computed efficiently. However, conversion from a chunk based structure to relational format can be particularly time consuming. To achieve greater efficiency, a comprehensive chunk based file organization is proposed for the data warehouse backend. In practice, backend storage may either be *natively chunked*, or based upon standard relational tuples that

Figure 2.13: Chunked Query

Figure 2.14: Chunked cube

Figure 2.15: 2-D Chunk

are re-arranged according to the chunked data format. In addition, a chunk index needs to be created for each chunk-file so that, given a chunk number, all the tuples associated with that chunk can be retrieved using the chunk index. We note that no such commercial system actually exists.

The solution proposed by Deshpande and Jeffrey Naughton is much more efficient relative to previous solutions. It is much better at minimizing the storage space required for the cache as it does not duplicate storage for overlapping query results. Such redundancy effectively reduces the size of the available memory for the cache and consequently reduces the hit ratio.

Since chunking can be done at different aggregation levels, the system has good closure properties, implying that chunks at lower levels can be used to obtain the result of the chunks at higher levels. Using Figure 2.15 as an example, we can see that if we want to find the CGPA for "Student A" in "Fall 08", we can simply combine the results of all the courses for "Student A" in "Fall 08" from Figure 2.14 to find the result for higher aggregation levels.

As promising as this solution is, however, a number of unresolved issues remain.

One of these is scalability. The MOLAP form of data storage generally works well in lower dimensions, but it falters in higher dimensions. Recall that the number of possible aggregation combinations for a d-dimensional space is exponential in the number of dimensions. So for our example in Figure 2.14, there would be $2^3 = 8$ different cubes. For a 10 dimensional space, there would be $2^{10} = 1024$ aggregation levels. More importantly, for an n dimensional MOLAP cube, with a cardinality set bounded as $K = (k_1, k_2, k_3.....k_n)$ for each of the dimensions, the size of the most detailed cube will be $S = \prod_{i=1}^{n} K_i$. For 5-dimensional cube, if all the dimensions have an average cardinality of 100, the size of the cube — and the potential search space — will be $100^5 = 10000000000$. While the cache only physically materializes part of this space, it is important to recognize that much of this volume will be extremely "sparse". In other words, there will be a great many null values in any given query region. For MOLAP-style storage or caching models, this can be extremely problematic as the representation of dead space greatly reduces the effectiveness of the cache.

A second issue with the chunk cache is that its authors suggest that optimal performance is achieved when the cache is large enough to store the entire base table/cuboid in memory. In fact a number of their experimental results exploit this feature. We find this assumption to be highly improbable in that data warehouse size generally scales into the Tera byte range. Moreover, caching the entire base table in memory would dramatically improve the performance of any caching system.

We conclude this section with a brief review of the pros and cons of the chunk based approach.

- **Advantages:**
  - It works very well with lower dimensions and lower cardinalities.

  - Since it is a MOLAP model, it has a natural indexing advantage.
  - It uses an aggregation technique that is very useful for aggregating from lower level hierarchies.

- **Disadvantages:**
  - Its bit-wise indexing scheme used for keeping track of each chuck at the group-by level is quite expensive.
  - It doesn't scale as well in higher dimensions, especially those with high cardinalities.
  - Full caching of the base cuboid is impractical.
  - An expensive conversion operation is needed during loading if the underlying data warehouse is a relational server.

2.6 Conclusions

In this chapter, we have systematically discussed much of the background information required to understand the research presented in the remainder of this thesis. We began with a general introduction to data warehouse systems, including their reliance on the common Star and Snowflake schema. We then discussed the importance of OLAP in the current data driven world. We described decision support systems and their main components, including the ROLAP and MOLAP physical models. We also looked at few of the more important research contributions in this area. We classified the solutions in terms of the degree of dynamic computation, as well as their physical architecture. Finally, we reviewed the chunk based caching of Deshpande and Naughton in some detail. As noted, we will return to this proposal later in the thesis.

Chapter 3

ROLAP based data warehouse caching

3.1 Introduction

In today's business world, decision support systems have become an integral part of the decision making and planning process. Given the central role of OLAP in DSS systems, researchers have become increasingly interested in the topic. As noted, however, the requirements of decision support systems are significantly different than those of OLTP systems, leading to both new constraints and new opportunities for those working on novel OLAP applications and architectures. This is certainly the case in the context of multidimensional OLAP caching. Some of the more obvious differences between OLAP and OLTP caches include:

1. The increased size of the cached objects
2. The multi-dimensional nature of the data
3. The requirement for multi-dimensional indexing support

Taken as a whole, this makes OLAP caching a potentially rich area for new research. Unfortunately, most of the research on OLAP caching thus far has concentrated on chunk-based caching (MOLAP) [10]. Not only are there limitations with this style of caching but, as previously noted, no existing relational databases natively support chunk-based disk storage. When ROLAP based caching has been examined, it has typically been done on a very small scale and with trivial test environments [32].

3.2 Motivation

In Chapter 2, we discussed two of the most significant contributions in the area of ROLAP/MOLAP caching. We noted that the work described in [32] was relatively limited in scope, particularly in terms of its inability to support the partial match paradigm so common in OLAP settings. Conversely, the research presented in [10] — while being quite significant in terms of its breadth — had serious scalability issues due to its reliance upon a MOLAP data structure that becomes increasingly sparse in more realistic OLAP spaces.

Given this scenario, we identify the following "high level" objectives for our new caching framework:

1. Provide support for both full match and partial match queries.
2. Design a tree structured cache/index that stores only those points actually found in the space.
3. Minimize the number of leaf nodes that must be searched.
4. Augment the basic structure with efficient cache management techniques.

In the following sections, we discuss how we approach the implementation of these target objectives.

3.3 The Sidera Parallel OLAP DBMS

In the following section, we will present a concise overview of the R^3-cache framework. However, since that system is in fact fully integrated into the larger Sidera OLAP DBMS, we begin with a brief description of the structure and function of the OLAP server itself. As previously noted, Sidera has been designed from the ground up as a parallel "shared nothing" platform for the resolution of complex multi-dimensional analytic queries. The current system consists of approximately 70,000 lines of C++ code and runs on a 17-node, 34-processor HP Proliant Linux cluster. Subsystems exist for data cube generation and distribution, as well as multi-dimensional selectivity estimation, OLAP indexing, and the manipulation of dimensional hierarchies. Concurrently running projects are extending the server with conceptual modeling facilities, high availability and fault tolerance features, and native language object oriented query interfaces. Figure 3.1 provides a simple illustration of the physical architectural framework. Note that Sidera is essentially constructed as a series of logically independent backend servers that are transparently bound together by a Parallel Service Interface (constructed on top of the Message Passing Interface). In essence, each server knows nothing of the existence of its sibling servers and operates solely on the cube fragments associated with its local resources. Each server contributes equally to the resolution of every individual query.

Each of the local servers also supports its own OLAP *stack*, of which the caching module is but one component. Figure 3.2 depicts the basic elements of the stack

Figure 3.1: The Sidera architecture.

and the relationship between them. Note that the caching subsystem sits below the query processor and hierarchy manager but above the low level indexing and storage components. In fact, both the cache and storage engine are oblivious to the hierarchies themselves and represent data solely at the base level of each dimension. It is the job of the hierarchy manager to transparently *map* final result sets — with the aid of the query processor — between arbitrary levels of the dimension hierarchies (i.e., roll-up and drill-down). This modular approach dramatically simplifies the processing logic of both the caching and indexing subsystems. A full discussion of the Hierarchy Manager is provided in [12]. Finally, we note that Sidera is designed specifically as an analytics server, and does not attempt to function as an "all things to all people" system. As such, detail-level ad hoc querying is (transparently) funneled to a local data warehouse *partner* (i.e, a commodity DBMS).

Figure 3.2: The OLAP stack.

3.4 Relational OLAP caching

Our new caching solution is physically based on a variation of the standard R-tree data structure. An R-tree, by design, is an indexing structure that allows rapid retrieval of selected records from disk based storage. In general, indexing mechanisms like the B+-tree and Hash Table have been core components of database management systems for decades. In multi-dimensional environments, mechanisms such as the B+-tree and the Hash have been extended in various ways to try to support queries on multiple attributes. Most of these methods have met with limited success but the R-tree has been shown to be one of the few natively multi-dimensional indexes that performs well in real-world environments.

As noted, however, the R-tree is traditionally associated with disk-based retrieval. A cache on the other hand is memory-resident. Nevertheless, it is important to understand that the ultimate purpose of an index is to provide efficient retrieval characteristics from a relatively large data store. OLAP caches represent just this type of environment. They are natively multi-dimensional and possibly quite large. Moreover, in order to provide the kind of performance that would be required from a cache, records must be located within the cache extremely quickly. For this reason, use of the R-tree as a fundamental component of a relational OLAP cache represents an attractive starting point for the design of our physical architecture.

3.4.1 R-tree

Classical, one-dimensional indexes are not particularly effective when it comes to searching a multi-dimensional spatial structure. Hash tables, for example, can be used to identify all spatial data points in a d-dimensional space (i.e., "exact location" queries). Unfortunately they are virtually worthless for the range queries commonly used in OLAP settings. Conversely, one may utilize a set of B+-trees to support searches of multiple attribute ranges [27, 4]. In this case, there are two problems. First, maintenance of all these indexes becomes quite expensive. Second, because the data can only be physically sorted on one of the B-tree dimensions, performance deteriorates rapidly when this dimension is not part of the user query.

The R-tree was first proposed as a native multi-dimensional index by Guttman in 1984 [17]. An R-tree is a height balanced structure that consists of two kinds of nodes — leaf nodes and non-leaf nodes. In an R-tree implementation, a node N corresponds to a disk page. If a disk page allows a maximum number M of entries in node N, then $m \leq M/2$ is the minimum number of entries permitted. In other words, nodes must be at least half-filled.

According to Guttman, the R-tree supports the following properties:

1. Every leaf node contains no less than m and and no more than M index records.
2. Every non-leaf node has no less than m and no more than M children boxes, unless it is the root.
3. For each index record in a leaf node (I, L), I is the smallest possible boundary that could hold these records.
4. For each child box in a non-leaf node entry (I, C), I is the smallest possible bounding box that can encapsulate the child boxes.
5. The root node must have at least two nodes.
6. All R-tree leaves are at the same depth.

We can use the notation $[I, L]$ to define a leaf node, with L pointing to the d-dimensional records and I identifying a d-dimensional bounding box for the leaf nodes. Further, I can be defined as $I = (I_0, I_1, I_2 ... I_d - 1)$, where d is the number of dimension and $I_i = [a, b]$, with a and b delimiting the boundary along the dimension i. Non-leaf nodes can be defined as $[I, C]$, where I identifies a d-dimensional bounding box and C contains the pointers to all the child boxes bounded by that node. If the data set has a point count of n and a minimum branching factor m, the height of the R-tree is bounded as $h = \lfloor \log_m n \rfloor$. Here, $m \leq M/2$, where M is the maximum number of points that can be stored in a node. This arrangement gives considerable flexibility to the R-tree. By changing the value of m, we can easily manipulate the height and thus the performance of the tree.

Figure 3.3 provides a simple illustration of the logical structure of the R-tree. Solid lines represent non-leaf nodes and dotted lines are leaf nodes. In this case, R5

Figure 3.3: Logical structure of the R-tree

to R10 are leaf nodes containing pointers to the data elements. R1 to R4 are non-leaf nodes housing pointers to the child boxes. As should be clear, nodes in an R-tree may intersect each other. The physical, tree-based structure corresponding to the logical model is provided in Figure 3.4.

Let us now look at a simple search on the R-tree. In Figure 3.3, the dark dotted line S identifies the search space. All points encapsulated by S must be returned as part of the query result. Note that the search space $S = (S_1, S_2, ...S_d)$ is a d-dimensional hyper-rectangular bounding box. S_i denotes the range along the i^{th} dimension, where $(1 \leq i \leq d)$. S_i can formally be represented as $S_i = [a, b]$, where a is the lowest value along the dimension i for search space S and b is the highest value along the dimension i. Let us denote the root node as R and non-leaf nodes as $[I, C]$, where I is a multi-dimensional bounding box and C is the set of pointers for the child nodes. Further let us denote leaf nodes as $[I, L]$, where I is a multi-dimensional bounding

Figure 3.4: Physical structure of R-tree

box and L is the pointers to the data blocks.

Searching the R-tree is similar to the process of searching any tree, with the major difference being that, since the boxes can overlap each other, more than one sub-tree may have to be searched. Algorithm 1 formalizes the logic of the search method. The traversal begins at the root and propagates down to the child boxes. If we take the R-tree example in Figure 3.3, Algorithm 1 initially finds that none of the child nodes of R1 intersects with S. So we move on to the next non-leaf node R2. Here, we find that leaf-nodes R8 and R9 intersects with S, so both boxes must be traversed to find the data points in user-defined region. Because of the possibility of overlap, it isn't possible to provide a meaningful lower bound on the search algorithm. In fact, post-Guttman variations of the R-trees have often focused on minimizing the degree of bounding box overlap.

Algorithm 1 Search algorithm

```
Input: $S = (S_1, S_2, ...S_d)$ is the search space containing the number of dimensions $d$
    $R$ = the root node
    $M = (M_1, M_2, ...M_d)$ is the set of all leaf nodes

Output: $L$, a set of leaf nodes

1: if $R \notin M$ then
2:    for each child-node $C$ in node $R$ do
3:       if $C \cap S \neq \emptyset$ then
4:          Search(C, S, M)
5:       end if
6:    end for
7: else
8:    if $C \cap S \neq \emptyset$ then
9:       Insert $C$ into $L$
10:   end if
11: end if
```

3.4.2 Non overlapping R-tree

Before describing the Non-overlapping R-tree, let us recap the purpose of our new R-tree design. Our primary goal is to build a very fast, in-memory OLAP caching system. It should of course be considerably faster than disk-based access and it should scale well to higher dimensions. Because the R-tree has proven itself to be quite effective in general purpose, multi-dimensional environments (i.e., disk-based), it made a great deal of sense to try to adapt the block-based disk model to page-based main memory. Given this target, one of the crucial objectives becomes the reduction of box overlap, particularly at the wide, lower level of the tree. We achieved this goal in the form of the NOR-tree (Non-overlapping R-tree).

Algorithm 2 describes how the search mechanism works for the NOR-tree. Simply put, when a new query arrives and intersects with an already existing query, we decompose the query into two sets. The first set contains $A_i \in A$, where $0 \leq i \leq d$ and $A = N \cap S$, with N the node to be searched and S the search query. The second set contains $B_i \in B$, where $0 \leq i \leq d$ and $B = (N \cup S) - S$, with N the node to be searched and S the search query. Results from the query set A are returned directly from the cache, while the query set B is sent to the backend data warehouse. Data returned from the data warehouse is inserted into the corresponding queries in query set B.

Algorithm 2 Search algorithm for NOR-tree

Input: $S = (S_1, S_2, ...S_d)$, is the search space containing d number of dimension.
$M = (M_1, M_2, ...M_d)$, is a set of all boxes.
$L = (L_1, L_2, ...L_d)$, is the set of newly created boxes.

Output: a set of data points

```
1: for each box $m$ in set $M$ do
2:   if $S \subseteq m$ then
3:     Return result from the box $m$
4:   end if
5:   if $S \cap m \neq \emptyset$ then
6:     insert $((S \cup m) - m) intoL$
7:     Return cached data in $(S \cap m)$
8:   end if
9: end for
10: if $L \equiv \emptyset$ then
11:   Return result for $S$ from database
12: else
13:   Return result for $L$ from database
14: end if
```

Figure 3.5 demonstrates how this would work with a simple example. In this case, "Query 1" has already been executed and its contents are now stored in the cache.

Figure 3.5: Query decomposition

When "Query 2" arrives, the cache manager determines a partial intersection with the "Query 1" object in the cache. "Query 2" is now decomposed into a pair of hyper-rectangular query sets, A and B. A is subsequently processed from the cache, while B is delivered to the disk backend.

Figure 3.6 concisely illustrates the difference between the regular R-tree and the Non-overlapping R-tree, in the context of a multi-dimensional caching system. In both models, we begin at the same point. In Stage 1, the R-tree is empty; it has no data points, child nodes, or bounding boxes. When the first query, "Query 1", is executed, the R-tree is searched and no match is found. Subsequently, "Query 1" is sent to the data warehouse. Query results returned from the data warehouse are then stored in the root node. At this point, the root node is also the leaf node. In Stage 2, when "Query 2" is executed, the root node is searched and we find that "Query 2" partially matches with the root node. The root node expands to hold both "Query

Figure 3.6: Non-overlapping R-tree

1" and "Query 2". At this stage, our data structure is essentially a standard R-tree in that it houses overlapping leaf nodes. In the final stage, however, overlapping leaf nodes in the NOR-tree are split into multiple leaf nodes (in this case, four). More importantly, none of the cache objects — Q1, Q2, Q3, or Q4 — share any portion of the space. Finally, we note that Figure 3.6 shows the most trivial example of a multi-dimensional search. In reality, the R^3-cache is far more complex. In the remainder of this section, we will discuss the process in greater detail.

3.4.3 Insertion and node partitioning

In this section, we describe how we populate the NOR-tree. Basically, this is a two-step process. In Step 1, we add records by traversing the tree to find the appropriate leaf node for insertion. Then, in Step 2, we evaluate the size of the leaf node and partition the node accordingly.

In general, our insertion mechanism is similar to regular R-tree insertion. We insert data points into leaf nodes. If the leaf node gets bigger than its maximum allowed size, we split it recursively into two sub-nodes. Then the split is propagated upwards. Figure 3.7 illustrates this process. At the beginning, root node A has a set of data points, $D = \{d_1, d_2, ...d_n\}$, with n being the number of points in the set and $n \leq$M, where M is the maximum number of points that can be stored in a leaf node. Next, we insert a set of data points, $D_{new} = \{d_1, d_2, ...d_p\}$ into D such that $M < (n + p) \leq 2M$. At the next stage, node A splits into two equal-sized leaf nodes, node B and node C, such that $1 \leq sizeof(B) \leq M$ and $sizeof(B) = sizeof(C)$. Thus we get two new leaf nodes and one non-leaf node. The following steps mirror the previous two steps for each of the newly created leaf nodes B and C.

Recursive partitioning is one of the most important elements of the NOR-tree approach. If the partitioning is badly skewed, it is likely that the R^3-cache will perform poorly. So it is important to provide a partitioning model that ensures that records close to each other in the multi-dimensional space remain mostly in the same leaf nodes. Our goal here is to reduce the number of intersections between the leaf nodes and incoming range queries since the number of intersections negatively impacts the R^3-cache performance. The reason for this is not only that increased intersections force the cache algorithm to traverse more boxes, but that intersections

Figure 3.7: Insertion and node splitting

have the potential to generate a large number of split queries which, in turn, reduces cache performance. Figure 3.8 is an example of a bad leaf node split. In this figure, we show a leaf node A with a slightly skewed data-set, which is not unusual for a data warehouse. A is subsequently split into a series of leaf nodes along the Y dimension. As a result of the inherent data skew, we get very bad splits. Next, in Figure C, we run "Query 1" over the leaf nodes to search for the cached data. Because of the less than ideal partitioning, "Query 1" has to be processed against all of the leaf nodes. Clearly, this makes the search quite inefficient.

Our partitioning approach, as illustrated in Figure 3.9, is similar to the mechanism proposed by Muralikrishna and DeWitt [26]. There, a two-dimensional equi-depth histogram called an *hTree* is used to partition the space. The model is much like that of a Grid-file in that the data space is recursively divided in half by using the value of one of the dimensions as a boundary at each step. The dimensions are chosen in sequence. In our approach, we also divide the data space recursively, each time in a different dimension. If our data space has dimension count d and $D = \{d_1, d_2, ...d_d\}$, where D is the set of dimensions in our data space, then the root node A will be partitioned along the sequence $\{d_1, d_2, ...d_d\}$. As Figure 3.9 shows, the generation of an equi-depth tree results in B rectangles, such that each rectangle encloses approximately $\frac{R}{B}$ points within it, where R represents the total number of points in the data space. As noted in [26], the sorting cost for the tree at each stage of the partition is $P * \log_b\left(\frac{P}{d}\right)$, with P representing the number of data pages, b denoting the number of partitions at each stage of the algorithm. This modest sorting cost lowers the construction cost and is particularly helpful during insertion and node-splitting since sorting is necessary at those stages.

Figure 3.8: Bad Partition

Figure 3.9: Good Partition

In terms of query resolution, the division of the grid in alternate dimensions removes the bias along a single dimension axis. This kind of bias tends to result in a cache that performs well for a small number of queries but very poorly for vast numbers of queries that do not benefit from the skewed partitioning. For example, in Figure 3.8, "Query 2" — depicted in D — is perfectly suited to the poorly partitioned cache. But for arbitrary queries of varying sizes and ranges, this cache will perform very poorly because of the large number of intersections/seeks produced. "Query 1" in C is such a case.

We note that the R^3-cache page is essentially constructed as a user-defined multiple of the OS page size. If the leaf-node size exceeds the current page size, leaves must be split, and encapsulated by new parent level bounding boxes. Again, this is similar to a conventional R-tree where a leaf level box is associated with a file system disk block. In general, techniques for R-tree splitting seek to minimize the total area of the constituent child boxes since unnecessarily large spatial divisions tend to encourage excessive seeks at query resolution time. For in-memory access structures, however, where seek time is irrelevant, our primary objective is to create a non-overlapping leaf space in which points are definitively stored in a single leaf node.

Algorithm 3 formally describes the algorithm for insertion and node-splitting.

3.4.4 Deletion

The deletion mechanism works by identifying the leaf nodes from which data points need to be removed. Once the data points are deleted from the leaves, the parent of the leaf nodes is identified. If the size of the parent node is less than M, then we move the data points of the child nodes to the parent node and remove all the children. We

Algorithm 3 Insertion

```
Input: $D_{new} = (d_1, d_2, ...d_n)$, is a set of newly inserted items
    $N$ is the leaf node, where $D_{new}$ will be inserted.
    $M$ is the maximum size of a leaf node.

1: insert $D_{new} intoN$
2: if $sizeof(N) > M$ then
3:   split the node to get new leaf-nodes $N1$ an $N2$
4:   if $sizeof(N1) > M$ then
5:     $Insert(\emptyset, N1, M)$
6:   end if
7:   if $sizeof(N2) > M$ then
8:     $Insert(\emptyset, N2, M)$
9:   end if
10: end if
```

designate the parent node as the new leaf and continue the same process up through the tree.

Figure 3.10 illustrates the full process. In this case, the NOR-tree has four leaf-nodes, D, E, F, and G. At first, n data points are removed from D and E, so that $sizeof(D) + sizeof(E) \leq M$, where M is the maximum size of a leaf-node. The data points $D_{remainingdata} \in D$ and $E_{remainingdata} \in E$ are inserted into B. Node D and E are then removed from the tree and B is designated as a leaf node. Similarly, both C and A are converted into leaf nodes.

Algorithm 4, and the supporting method described in Algorithm 5, formally describe the deletion process.

3.4.5 Complicating factors

One of the problems with the general partitioning model — at least as we have described it thus far — is that it has the potential to create an exponential number

Figure 3.10: Deletion

Algorithm 4 Deletion

```
Input: $P = (p_1, p_2, ...p_n)$ is a set of non-leaf-nodes
    $I = (i_1, i_2, ...i_n)$ is a set of items to be deleted
    $L = (l_1, l_2, ...l_n)$ is the set of leaf nodes
    $M$ is the maximum size of a leaf node.

1:  $L = FindLeafNodes(I)$
2:  for each leaf-node $l_i$ in set $L$ do
3:      $Remove(I, l_i)$
4:      $p_i = FindParentNode(l)$
5:      if $p_i \notin P$ then
6:          $Insertp_i inP$
7:      end if
8:  end for
9:  for each non-leaf-node $p_i$ in set $P$ do
10:     $CHECKPARENT(p_i)$
11: end for
```

Algorithm 5 *CHECKPARENT*

```
Input: $P$ = is the non-leaf-node
    $size_{total} = 0$

for each child-node $l_i$ in $P$ do
    $size_{total} = size_{total} + sizeof(l_i)$
end for
if $size_{total} \leq M$ then
    for each child-node $l_i$ in $P$ do
        insert all data points, $d \in l_i$ into $P$
        Remove $l_i$
    end for
    Set $p_i$ as a leaf-node
    $CHECKPARENT(Parent(P))$
end if
```

of new cache elements. In the worst case, when a new query completely subsumes an existing node, it can generate $O(d^3)$ new queries, where d is the number of dimensions. So, for example, for a two-dimensional data structure, it might generate $2^3 = 8$ new queries that need to be retrieved from the data warehouse. Figure 3.11(B) illustrates the problem. Here, a new query in the two-dimensional space subsumes the existing node N. After the split process, $2^3 = 8$ new backend queries are generated. For a three-dimensional space this number would be as high as $3^3 = 27$, and for a four-dimensional space $4^3 = 64$. These numbers are simply unsustainable in caching scenarios.

To address this problem we devised the solution of merging adjacent query elements. Let us define d as the number of dimensions for queries A and B. Further, the set of ranges $R_A = (R_{A^1}, R_{A^2}, ...R_{A^d})$ defines the range-set for A, while the set of ranges $R_B = (R_{B^1}, R_{B^2}, ...R_{B^d})$ defines the range-set for B. For query A and query B to be merged, we need to satisfy the following criteria. For $j = d - 1$, $R_{A^i} \equiv R_{B^i}$

Figure 3.11: Query merging

must hold for the dimension count j, where $1 \leq i \leq d$, and $R_{A^k} \neq R_{B^k}$ must hold true for just one dimension k, which can be any dimension between 1 to d. In addition, we must have $R_{A^{k^1}} >= R_{B^{k^0}} - 1$ and $R_{A^{k^1}} <= R_{B^{k^1}}$ and $R_{A^{k^0}} < R_{B^{k^0}}$, where $R_{A^{k^0}}$ is the lower end for dimension R_{A^k}, $R_{A^{k^1}}$ is the higher end for dimension R_{A^k}, $R_{B^{k^0}}$ is the lower end for dimension R_{B^k}, $R_{B^{k^1}}$ is the higher end for dimension R_{B^k}. In other words, for two 3-dimensional queries to be merged they must satisfy these two following criteria. First, 2 of their dimensions must be equal to one another. Second, for the other dimension, one of the queries *highervalue* in that dimension must be between the *lowervalue* $- 1$ and *highervalue* of the other query. In addition, its *lowervalue* must be less than *lowervalue* of the other query.

Our solution solves the problem by reducing the number of new queries from $O(d^3)$ to $O(2d)$, as demonstrated in Figure 3.11(C). A somewhat more realistic example is provided in Figure 3.12. Given Scenario 1, we see that the query box is totally subsumed by the cache node. This is clearly the best scenario. In this case, no new query boxes are generated and all results are returned from the cache. With Scenario 2, the query box now only partially intersects with the cache node. Only one edge of the X-axis and Y-axis from the query box is within the area of the cache node. This results in the generation of only three new boxes. We now examine Scenario 3. Here, the query box partially intersects with the cache node. Both edges of the X-axis and one edge of the Y-axis from the query box is within the area of the cache node. This results in the generation of five new boxes. Finally, Scenario 4 generates the largest number of new boxes since the entire cache node is inside the query box. Formally, the number of new boxes in the worst case is exactly $d^3 - d + 2$, where d is the number of dimensions. For a two-dimensional cache, the worst case scenario

Figure 3.12: Different query scenarios

generates $2^3 - 2 + 2 = 8$ new boxes. For a three–dimensional cache, the worst case scenario would be $3^3 - 3 + 2 = 26$ new boxes.

In Figure 3.13, we demonstrate how we would get 26 new boxes in the worst case scenario for a three-dimensional space. Specifically, (a) shows a cache node that is completely subsumed by the query box. In the subsequent figures we describe how we split the query box to get 26 new query boxes. (b) shows the front end of the newly generated query boxes. There are nine boxes contained in this set. (c) contains a set of boxes that are exact replica of the boxes contained in (b). (d) shows the position of the query sets contained in (b) and (c), relative to the cache nodes. As indicated, (b) and (c) are located opposite to each other, with the cache node in the center. Similarly, the remaining query spaces are split into two halves — first by the left-hand and right-hand sections and finally top and bottom. The reason for this unusual split has to do with the position of the cache node in the center of

Figure 3.13: Worst case scenario for 3-D Cache

the query box. (b) and (c) generated nine boxes each. (e) and (f) generate three boxes each. (h) and (i) generates a box each. Combining these boxes together, we get $(9 \times 2) + (3 \times 2) + (1 \times 2) = 26$ new boxes. We note that a similar pattern unfolds in higher dimensions.

As indicated above, our query merging solution reduces the number of new queries from $O\left(d^3\right)$ to $O\left(2d\right)$. As such, we should have only $2 \times 3 = 6$ new query boxes for the previous example. In Figure 3.14, we show how we can reduce the number of query

Figure 3.14: Query merging in 3-D cache

boxes to six by merging adjacent boxes. In (b), we have nine adjacent boxes. We can therefore combine them and reduce the number of boxes to one. Similarly in (c), we can combine the nine adjacent boxes and reduce them to one. Likewise in (d) and (e), we merge the adjacent boxes to reduce six new boxes to only two. Finally, (h) and (i) are single boxes themselves, so there is nothing to merge. Counting all these merged boxes give us a total of six news boxes. By extension, a similar manipulation of a four-dimensional cache space would produce at most $2 \times 4 = 8$ new boxes.

We conclude this section with a more concrete discussion of the worst case bounds on node splitting. We begin with the original partitioning process, which we indicated had a worst case of $d^3 - d + 2$. The splitting process works in three steps. In the first step, it splits the box in any one direction. This creates $d - 1$ sets, with each set consisting of d^2 adjacent boxes, where d is the number of dimensions. In Figure 3.13, (b) and (c) represent these sets. This first step creates $d^2 \times (d - 1)$ boxes. During the second step, the remaining area of the boxes are split again. But this time the process creates $d - 1$ sets, each set consisting of d adjacent boxes. In Figure 3.13, (e) and (f) represent these sets. This second step creates $d \times (d - 1)$ boxes. After the first two steps of splitting, there will always be two boxes remaining. Combining all these boxes gives us $d^2 \times (d - 1) + d \times (d - 1) + 2$ boxes. Now, $d^2 \times (d - 1) + d \times (d - 1) + 2 \Rightarrow d^3 - d^2 + d^2 - d + 2 \Rightarrow d^3 - d + 2$.

Now, let's take the equation $d^2 \times (d - 1) + d \times (d - 1) + 2$ and try implementing our new query merging solution. Since our query merge combines the adjacent boxes into a single box, this equation can be rewritten simply as $1 \times (d - 1) + 1 \times (d - 1) + 2$, which gives us $2d$.

3.5 Cache management

For every data warehouse caching system, cache management is a vital part of the solution. Because the queries in the warehouse are complex in nature and potentially very large, it is important to have a caching framework that is fast, efficient and reliable. Despite having a good underlying architecture, a caching system can degenerate into a slow, inefficient bottleneck if proper methods are not followed to maintain its contents.

Managing the cache system is an integral part of our proposed solution. In this section, we present two algorithms, one for cache object replacement and one for admission into the cache system. We also describe how we define "hot spots" within the cache and exploit low-use periods to fill up the space in the hot areas. The general solution makes extensive use of meta data stored in the tree nodes. These meta-data values include query size, frequency, and query response time. Given its importance to the replacement and admission algorithms, we begin with a discussion of the meta-data itself.

3.5.1 Meta-data

To effectively maintain the cache, we need node-related information to be stored in a systematic fashion. Instead of creating a separate data structure, as was done with the lattice described in [10], we store information directly within the tree nodes. Doing so avoids a partial replication of the tree, which would have minimized the amount of memory available to the cache. We describe our meta-data values below.

- **Query size**: The size of the query result-set is in bytes. For a leaf node, this will be the space associated with all data points stored within it is boundary. For non-leaf nodes, this will be the *combined size* of all data points stored within its leaf-nodes. This information is gathered when the node is created. We denote the Query Size as S in this section.

- **Frequency**: The frequency with which this cache is accessed. The Frequency F can be defined as $F = \frac{T}{N}$, where T = amount of time (in seconds) that the node has resided in the cache, and N is the number of times the node has been accessed. Parent nodes are used to measure frequency for the nodes that have no previous records through which frequency can be measured. The Frequency value is updated each time the cache is accessed.

- **Query response time**: The estimated time it will take for the particular node to be queried from the actual data warehouse. It is calculated from the time the query is sent to the data-warehouse to the time query results are returned from the data warehouse. If a query q is sent to the database at time $t1$ and the database returns the query result at time $t2$, query response time will be calculated as $T = t2 - t1$. This information is particularly useful because it helps the system reduce query response time by caching the nodes with high query response time. Query response time is gathered when the query is physically executed in the data warehouse. We will denote the Query Response Time as T in this section.

- **Cache value**: This is used to decide whether a node is cache-able or replace-able. The Cache Value is derived from a formula that uses the three previous parameters. Its exact format will be discussed shortly. This value is updated each time any of the above parameters are updated. The reason for keeping the Cache Value with the node is to reduce the execution time during cache processing. We'll denote the Cache Value as V in this section.

Figure 3.15 illustrates the process by which meta-data is stored and updated as the nodes are accessed. At Stage 1, the user sends a query Q. Since the cache does not have any data required by query Q, the query is sent to the data warehouse at time $t1$. The data warehouse returns the query results of size S at time $t2$. At this point, the query size is S and the query response time $T = t2 - t1$. Now, we measure frequency by the ratio between amount of time the node resided in the cache and the number of hits for that node. If we take current time as $t3$, then the amount of node resided in the cache will be $t3 - t2$. So, Frequency $F = \frac{t3-t2}{1}$. At stage 2, query size and query response time remains the same as before. However, since the number of node accesses N and the amount of cache time T changes, frequency $F = \frac{t4-t2}{2}$ changes with them too.

Figure 3.16 shows the meta-data relationship that exists between parent and child nodes. In this figure, P is a parent node, $P1$ and $P2$ are the child leaf nodes. Parent node P itself does not house any data, so its meta-data is calculated using the meta-data information of its child nodes.

3.5.2 Hotspots

Detecting hotspots in the R^3-cache was one of our major challenges. By a hotspot, we mean a high density area of the cache that is accessed very frequently by incoming queries. We use the Cache Value of a node to detect the hotspot areas within the cache. Generally, non-leaf nodes with the highest cache values are designated as hotspots. These hotspots are consequently used to locate nodes for cache replacement and cache updating during less active times. In Figure 3.17, we designate the R3 sub-tree as a hot-spot area of the cache because the node R3 holds the highest cache value amongst all the nodes.

Sometimes a designated hotspot area may grow too large to be easily managed by the cache. For example, when we try to pre-fetch data for hot-spot area R3 from the data-warehouse, the resultant data set is inadmissible to the cache because of its large size. To counter this situation, we can drill-down to the child nodes. Essentially,

Figure 3.15: Meta-data for cache management

Figure 3.16: Relationship between parent and child nodes in terms of meta-data

we identify the non-leaf child nodes with the highest cache-value to be the designated hot-spots. This process is repeated recursively until a desired hot-spot can be found. For example, in Figure 3.17, R7 could be identified as the next hot-spot since it holds the maximum cache value amongst all the child-nodes.

Algorithm 6 describes the mechanism used to locate the node with the highest cache value. It uses a standard depth-first search method to find the node. Usually the hot-spot found using this approach resides very high in the tree because the large nodes at the top of the tree have the highest cache value. In Algorithm 7, we refine the previous algorithm further to locate the hotspot areas more precisely. After locating the initial hotspot node V with Algorithm 6, we pass V as parameter to the $RefineHotSpot$ function. $RefineHotSpot$ finds the child node with the highest cache-value and recursively moves downwards until it reaches the lowest non-leaf level. The algorithm terminates when it finds a node containing a child leaf-node.

Algorithm 6 LocateHotspot

```
Input: $C_i$ = The non-leaf-child-node with the highest Cache Value in $i^{th}$ iteration
    $NodeWithHighestCacheValue$ = Highest cache value found so far
    $V$ = current node

1: for each non-leaf-child-node $V_i$ in $V$ do
2:   $C_i = LocateHotspot(V_i)$
3:   if $(CacheValue(C_i) > CacheValue(C_{i-1}))$ then
4:     $NodeWithHighestCacheValue = CacheValue(C_i)$
5:   else
6:     $NodeWithHighestCacheValue = CacheValue(C_{i-1})$
7:   end if
8: end for
9: RETURN $NodeWithHighestCacheValue$
```

Figure 3.17: R^3-tree and the meta-data

Algorithm 7 RefineHotspot

Input: V = current node

```
1: Find child-node $V_i$ with the highest cache-value
2: if $(Child(V_i) = leaf - node)$ then
3:     RETURN $V_i$
4: else
5:     RETURN $RefineHotspot(V_i)$
6: end if
```

3.5.3 Cache management policy

This section defines the policies that guide the cache management activities. In particular, we look at the criteria used for replacement and admission of a node in the cache. We also define the hotspot criteria we use for pre-fetching data during the inactive cache periods.

Cache replacement

Cache replacement for database management systems has been extensively studied [21, 29, 34]. In general, the idea behind these cache replacement techniques is to maximize the hit-ratio of the cache. That being said, little or no attention has been paid to minimizing the cost of querying from the disk backend. But data warehouse caches differ from those in operational databases in a significant way. Operational cache concern themselves with exact match retrievals. In other words, because there is no notion of a multi-dimensional space, the concept of intersecting partial matches makes no sense. This is a significant new opportunity for those designing new DW/OLAP cache. However, it is very important to note that due to the enormous size of many data warehouse queries, even small misses can eliminate most of the benefit of partial match resolution. Put another way, if a DW/OLAP query obtains 90% of its data from the cache, there may be little performance benefit if the remaining 10% of the data has to be retrieved from a very slow disk.

Given this fact, any cache replacement policy for data warehouses and/or OLAP servers must include the reduction of disk access time as one of its prime objectives. The WATCHMAN solution proposed by Scheuermann et al. [28] makes some progress in this regard. Still, their solution is quite generic in nature and is not ideally suited to the R^3-cache architecture. For example, our own system benefits when the size of the cache node is increased. Since larger cache nodes store more data points, traversal and split costs can be reduced. This is something the WATCHMAN approach could not exploit. Nevertheless, we are able to borrow some of the fundamental ideas from the earlier paper and adapt them to our own requirements.

To begin, we define the Cache Value $V = (F + T + S)$. Before actually calculating V for a given object, we normalize the values of F, T and S, so they range between 0 and 1. We do this with the meta-values since any one variable might otherwise have an extreme impact upon the calculation of V. This would result in the inappropriate influence of this one item of meta-data.

Now, as the cache value V gets lower, the probability for the node to be replaced gets higher. The principle behind this formula is that we would like to cache a node which has a higher frequency, lower query response time, and larger size. Thus, the chance of a node getting cached is proportional to the frequency, query response time and query size. So we can conclude that $V \propto F$, $V \propto T$ and $V \propto S$.

In Algorithm 8, we describe the cache replacement mechanism. *size* is the amount of space needed to be freed. In each iteration of the algorithm we find the non-leaf node with the i^{th}-least cache value and insert it in set C. When the combined size of all the nodes in set C becomes greater or equal to that of $size$, we return set C to the calling function. There are a few reasons for excluding the leaf nodes from the set. Leaf nodes are small and limited in size. Thus, compared to the non-leaf nodes, they are in a disadvantaged position, as shown in Figure 3.17. Another reason is that removing a small leaf node from a set of adjacent leaf-nodes creates a gap in the parent non-leaf nodes that, in turn, creates the potential to generate many split nodes in a comparatively small space. Later, we will discuss how we can identify hotspots in the R^3-cache and pre-fetch data during inactive periods in order to fill-up these holes in the cache.

Algorithm 8 Cache replacement algorithm

```
Input: $size$ = space needed to insert new queries into the cache
    $NODECOUNT$ = number of the non-leaf nodes in the R-tree,
    $C_i$ = non-leaf node with the least Cached Value,
    $C$ = set of non-leaf nodes to be returned,
    $node_{size}$ = combined size of the non-leaf nodes to be returned

1: for $i = 0$ to $NODECOUNT$ do
2:   $C_i$ = FIND non-leaf node with $i^{th}$ least Cache Value
3:   Insert $C_i$ into $C$
4:   $node_{size} = node_{size} + sizeof(C_i)$
5:   if $(node_{size} \geq size)$ then
6:     RETURN $C$
7:   end if
8: end for
```

In Figure 3.17, we presented a small scale R-tree and its corresponding meta-data. The first image shows the logical cache structure — the way cache nodes are placed related to one another. The second image shows the tree structure of the cache — how the parent-child nodes are connected to each other. This image also shows the meta-data stored in some of the selected nodes. Close observation of this data shows us that parent nodes are basically the product of their child nodes. For example, meta-data in the non-leaf node R7 is the product of its child nodes R7.1, R7.2, R7.3 and R7.4. This meta-data also helps us explain the decision to replace a node. If we assume this R-tree is full — meaning the cache-size has reached its maximum size limit — we have to select a set of low-performing nodes as candidates to be replaced. As we have described earlier, the non-leaf nodes with the smallest cache values are selected for replacement. Here, R10 and R11, with cache values of 1.30 and 1.20, are the prime candidates for replacement.

Cache admission

For any other caching platform, a Cache Replacement policy alone is sufficient. But for data warehouses, it is vital to verify the pages to be replaced and whether the inserted nodes are admissible to the cache. Algorithm 9 describes the logic behind our cache admission policy. If the result sets from the new query are $R = (r_1, r_2, .., r_n)$, then the required size for the new results sets will be $RS = \sum_{i=1}^{n} sizeof\,(r_i)$. If $RS > ES$, where ES is the remaining space in the cache, then the cache replacement algorithm is called. The replacement algorithm provides a set of nodes $C = (c_1, c_2, .., c_n)$ with least *Cached Value*, such that the size of the returned node size will be $RCS = \sum_{i=1}^{n} sizeof\,(c_i)$, where $RCS \geq RC$. Now, for each of the result sets $r_i \in R$, if $\exists\,(CacheValue\,(c_i)) \leq CacheValue\,(r_i)$, then r_i is inserted into the cache. Otherwise, r_i is not permitted into the cache. At the end of this process, if the entire result set R is admitted into the cache, then the set C is removed entirely from the cache. If q entries from R are not permitted into the cache, then the top q entries from C with the highest Cache Value are left in the cache and the rest are removed.

Algorithm 9 Cache admission algorithm

```
Input: $size$ = space needed to insert new queries into cache
    $C$ = set of nodes with Least Cache Value
    $R$ = set of query results to be inserted into Cache
    $node_{count}$ = number of nodes in $C$

1: $C$ = Cache replacement Algorithm($size$)
2: $node_{count} = sizeof\ (C)$
3: $count = 0$
4: for each result $r_i$ in result-set $R$ do
5:   $bool$ = false
6:   for each node $c_i$ in set $C$ do
7:     if CACHEVALUE($r_i$) $\geq$CACHEVALUE($c_i$) then
8:       $bool$ = true
9:     end if
10:  end for
11:  if $bool$ = false then
12:    $count = count + 1$
13:    remove $r_i$
14:  end if
15: end for
16: for $i = 0$ to $node_{count} - count$ do
17:   remove node with $i^{th}$ least Cache Value
18: end for
```

In Figure 3.18, we illustrate the cache admission and replacement process. We begin with the same logical cache structure we described in Figure 3.17. Now, the first image of this figure shows a new query Q overlapping box R7. The second image shows a newly created leaf node. Let us assume the cache value of this new node to be $F + T + S = 1.5$. Given the absence of any past history, we use the frequency value of the parent node as the new node's frequency value. So, in this case, we use the frequency value of the parent node R7 for our new node. For the new node to be admissible in the cache, two criteria must be met first.

First, the cache must have enough space for the new node to be inserted. If necessary, we need to replace low-performing nodes so that they make enough space for the new nodes. As we have mentioned before, our assumption is that the cache is full. We also previously identified R10 and R11 as the leading candidates to be replaced. Let us assume the real size of the new node to be 1500 bytes and the normalized cache value to be 0.9. So, for the new node to be admissible in the cache, non-leaf nodes with a size of at least 1500 bytes must be replaced first. Let us also assume that R10 and R11 have a combined size of 1600 bytes, which leaves enough space for the new node to be placed in the cache.

Second, new nodes which are to replace the existing nodes must have a cache value greater or equal to all the nodes to be replaced. If any of the new nodes fails to pass the threshold, then that node will be inadmissible. In this figure, the cache value for the new node is 1.5, which is larger than both R10 and R11. If any of the chosen nodes which are to be replaced had a larger cache value, the new node would have been inadmissible.

Since both the criteria are met in this case, we remove nodes R10 and R11 along with all their child nodes. Since R2 is now left with no child nodes, R2 is deleted too. The third image from Figure 3.18 illustrates this process. Finally, the fourth image shows that the new node R10 is inserted into a newly expanded node R7. As a result of the R7 expansion, its parent node R3 is also expanded. R10 is split into two halves because the size of the node is greater than $1K = 1024$ bytes.

Figure 3.18: Cache admission and replacement

Hot spot pre-fetching

As we have mentioned previously, DW/OLAP queries normally involve a small number of large queries. So, even if the queries themselves are more complex — which is why we use dedicated OLAP servers — they are a lot less frequent than operational database queries. In other words, there is usually a time lag between two consecutive queries. This "window" gives data warehouse designers an interesting opportunity to experiment with query resolution models. In our own research, we try to exploit this window by *pre-fetching* data from the backend database into our cache. The concept of data pre-fetching is not new and has been explored in various hardware and/or software environments [26, 14, 23, 25, 30]. In each case, the goal is the same — to improve performance by increasing the likelihood of a cache hit. In our case, the benefit is twofold.

1. By pre-fetching data from the database, we increase the cache-hit ratio and reduce the query execution time.
2. By filling up the empty spaces within the cache, we reduce the likelihood of creating a large number of small nodes. This, in turns, helps the cache to be more balanced and efficient.

Of course, pre-fetching data is not very useful by itself (or might even be counter-productive) if the pre-fetched data has a low probability of being queried. For this reason, it is important to fetch only those data points that have a high probability of eventual query access. The cache value is very useful in this regard. Specifically, we can use the cache value to locate hotspots in the cache and then pre-fetch data to fill up the empty spaces.

Figure 3.19 provides an illustration of the process by which we do this. In Section 3.5.2, we described how we detect a hotspot area of the cache. Initially, we identified R3 as a potential hotspot. We further refined our search and located R7. Again, our goal here is to fill up any empty spaces within the hotspot area. As we can see in Figure 3.19, A and B are the two empty areas within the node R7. So we define two *cache-generated* queries containing the areas A and B and sent them to the data warehouse. Again, this would be executed as a background process while no user queries are in the execution queue. If there is enough space available in the cache to store the data sets from the resultant queries, we create two new nodes, R7.7 and R7.8, and insert the data points accordingly. The third image shows the benefit of our query merging technique. We merge nodes R7.5 and R7.7 to create R7.5, and R7.6 and R7.8 to create R7.6, since both of the new boxes satisfy the condition described in Section 3.4.5.

Let us look at a graphical representation of the effectiveness of pre-fetching. Figure 3.20 starts with the same R-tree as the previous figure but without the pre-fetched data. The first image shows a query Q overlapping the nodes R7.3, R7.4, R7.5, R7.6, and two empty areas. The resultant R-tree shows two new nodes R7.7 and R7.8. Not only did these two nodes result in a cumbersome R-tree that has the potential to create many small nodes along the narrow edges, but their results had to be fetched from the backend database, which reduces cache efficiency. If we compare this outcome to that of the second image, where the R-tree consists of pre-fetched data, we see that the cache does not have to create any new nodes or fetch any data from the disk. Thus, a significant increase in efficiency can be realized.

Figure 3.19: Cache pre-fetching during inactive time

Figure 3.20: Comparison of R-tree with pre-fetching and without pre-fetching

3.6 Review

In Section 3.2, we identified several critical objectives that we needed to achieve. In this section, we briefly review our accomplishments.

1. ***Designing an efficient and effective data structure.***

   By modeling our data-structure as a R-tree, we have been able to effectively handle the multi-dimensional nature of our problem space. The R-tree is one of the most efficient data structures when it comes to dealing with spatial environments like ours. Moreover, we designed non-overlapping leaf nodes, using the NOR-tree, to make searching and updating of records more efficient.

2. ***Minimizing the number of leaf nodes.***

   We presented a number of techniques to reduce or minimize leaf node generation. The first dealt with query merging. Using $2D$ and $3D$ cube diagrams, we demonstrated how to reduce the number of leaf nodes from $O(d^3)$ to $O(2d)$ in the worst-case. We also developed a technique to pre-fetch data from the data warehouse to fill up the empty spaces in the hotspot areas and, if possible, to merge them with existing nodes. This technique reduces the number of small backend queries by making sure there are no empty spaces in the box.

3. ***Design adequate cache management policies.***

   Like all cache systems, cache management plays a large role in the efficiency of our system. We identified various meta-data elements and defined the meta-data relationships between the parents and their child nodes. We calculate a "cache value" for each node and used this cache value to identify the hotspot area, find the low-performing cache objects, and calculate the admissibility of the new nodes into the cache. We also described how to make use of inactive periods by pre-fetching data from the database to fill up the empty spaces in the cache.

3.7 Conclusions

In this chapter, we have presented the ROLAP based R^3-cache. The fundamental data structure of the R^3-cache is largely based on the R-tree. That said, it differs from the R-tree in the structure of its leaf nodes. Whereas the leaf nodes of a conventional R-tree may overlap, our R^3-cache eliminates overlapping nodes at the leaf level. Though this makes our cache system more efficient, at the same time it presents us with a new challenge of reducing the potentially large number of leaf nodes that are subsequently generated. We solved this problem by employing the techniques of query merging and hotspot pre-fetching.

As discussed in Section 3.2, some of the more significant previous cache research uses MOLAP-based techniques. Since the underlying data warehouse framework (Sidera) used for this research is based on ROLAP, we needed to develop a ROLAP cache system to ensure seamless integration with the parent architecture. Unfortunately, the few ROLAP based data warehouse caching frameworks that have been proposed aren't large enough in their scope to deal with a system as big as Sidera. We believe the caching model that we are proposing in this research makes a significant contribution to the literature in this respect.

Chapter 4

Evaluation

4.1 Introduction

In the previous chapter we fully discussed our proposed caching architecture. We described the design, algorithms, and policies employed by the R^3-cache. In this chapter, we give complete details of the experiments we have run on our system. We describe the test environment, including the various parameters used during the evaluation. We define the experimental data and provide analysis of these test sets. We also provide a comparative analysis with an alternative caching framework. Specifically, we contrast our system with the MOLAP framework proposed in [10], which we described in detail in the previous chapter. It's important to note that we have implemented a fully functional version of the MOLAP system in order to provide a fair and balanced comparison with our own architecture. We will discuss the comparative analysis in more detail in Section 4.4.

This chapter is organized as follows. In Section 4.2, we describe the platform on which the experiments were conducted. We specifically mention the hardware and software used for the evaluation. In Section 4.3, we describe the test framework. We explain the parameters used for the experiments and provide the ranges of these parameters. We also briefly describe the tools that we used to generate the experimental data. In Section 4.4, we provide the actual results, as well as the associated analysis. We provide a comparative analysis with the aforementioned MOLAP model. In Section 4.5, we conclude the chapter with a short review.

4.2 Physical platform

In this section, we give a brief description of the platform used for our experiments. For the sake of consistency, we use the same physical framework for both our own cache and the comparative system.

The following software components were used for performing experiments.

- **Operating system:** Fedora Core 5 (Red Hat Linux distribution)
- **Programming Language:** GNU C
  C++ Compiler version 4.2.0
- **Parallel processing:** LAM (Local Area MultiComputer)/MPI version 7.1.3

The hardware consisted of the following components:

- **RAM:** 1 GB RAM
- **Processor:** Intel Pentium(R) 4 CPU 3.00GHz
- **Hard-drive:** 140 GB ATA disk drive

Beyond this, it's important to note that although this system is built to run within the parallel Sidera system, our caching framework can be most effectively evaluated by limiting the system to just a single backend node. Recall that while Sidera consists of p nodes, each of these sibling servers is essentially identical. Moreover, each maintains its own independent cache that is backed by its own local OLAP disk subsystem. Therefore, interpretation of the experimental results is simply more intuitive if we restrict ourselves to just a single cache.

4.3 The test environment

We use synthetic data sets exclusively for our test-case scenarios. Synthetic data sets provide us with the flexibility to easily and effectively modify our test cases. It opens up possibilities for testing the system from many different angles, using many different parameters. We note that while real-world data sets are often used in database test environments [6], they tend in general to be either surprisingly uniform or one-dimensional in terms of what they can tell you about the systems being evaluated.

In data warehouse settings, caches tends to be very skewed in nature, if for no other reason than that user query tends to be very biased in nature. In particular, most users tend to iteratively query in one specific region of the space, to the exclusion of other regions. This user behavior is reflected in the state of the data warehouse caches, as they quickly come to favor a small number of key areas. To more accurately reflect the real-world nature of data warehouse caches, we start by generating skewed data sets, using a data set generator written specifically for Sidera's multi-dimensional spaces. The generator takes a variety of parameters including record count, dimension count, dimension cardinalities, and a zipfian data skew measure [37]. In the current case, we set the zipfian skew value in the range from 0.8 to 1.0, thereby giving fairly pronounced skew patterns.

To simulate the biased behavior of arbitrary users when submitting queries, we first generate a batch of 600 queries. The first 300 of these queries are totally random (on range and dimension), which helps to build up the initial cache contents. The next 300 queries represent variations of many of the same queries executed in the first block of queries. These queries are generated by a random query generator called qGen which was again built in-house for the specific purpose of testing the Sidera platform.

We tested with data sets ranging in size from 100,000 tuples to 10,000,000 tuples with a maximum 10% cache limit. In other words, the cache was limited in size to at most 10% of the size of the underlying database. Since dimension count plays a major role in distinguishing the performance difference between MOLAP and ROLAP systems, we ran tests using dimension counts between two to seven. We also used cardinality ranges of 1 to 1000 to generate our data sets. We used large cardinalities in particular to assess the performance of MOLAP systems, as these models tends to deteriorate in this setting.

4.4 Experimental results

We begin this section by comparing results between the MOLAP-based solution [10] and our own ROLAP-based model. We start with two batches of data sets. The first batch is composed of sets with one million tuples, while the second batch houses sets of ten million tuples. Both the batches are skewed with a maximum zipfian value of 1.0. Each batch contains six data sets with dimensions ranging from two to seven, the most common values in real world OLAP settings. As illustrated in Figure 4.1, while the performance for the ROLAP cache improves with the higher dimension count, the performance for MOLAP[13] on the other hand deteriorates with the higher number of dimensions. As we can see, initially for a dimension count of two, MOLAP actually performs better than the ROLAP cache. But with increased dimensions, we see the trend moving in the opposite direction. By seven dimensions the difference in performance reaches a factor of 4–5. We can attribute the superior performance of the ROLAP cache in higher dimension to two things.

First, our cache management policy works much better in higher dimensions as the frequency rate begins to decline. Higher frequency values occur in lower dimensions since there is little room to maneuver in this space. Consequently, a high number of unintended boxes gets flagged as frequently hit. Conversely, cache efficiency improves — with respect to the use of the frequency rate — with an increase in the number of dimensions. Since the MOLAP system doesn't have an advanced cache management policy, it is very limited in how it is able to manipulate the cache in higher dimensions.

Second, lower dimension queries tend to return higher numbers of records, in that data is compressed into a smaller area. Queries in larger dimensions reduce the portion of the region to be searched. Data also tends to be more thinly distributed. Because ROLAP cache objects only contain points that actually exist, it is well suited to increased dimensionality. With MOLAP, however, this is not the case since, in the absence of a complex array compression scheme, complete sub-grids must be represented. So while an increased number of records does not significantly impact MOLAP performance, the number of dimensions does.

In Figure 4.2, we present a similar analysis with different data sets. In this case, we leave most data set parameters the same, except that we reduce the zipfian value to 0.1. This produces relatively uniform data. The goal of this experiment is to show

Figure 4.1: Comparative analysis with MOLAP system for (a) number of records = 1 million and (b) number of records = 10 million

that our system has superior performance with data sets ranging from relatively uniform to the most skewed. As was the case with previous experiments with highly skewed data sets, our cache performs just as well with a uniform data set. In this case, there is relatively little difference at two dimensions, but the ROLAP cache improves performance by about a factor of four in higher dimensions.

In Figure 4.3, we used the same data sets as the ones in Figure 4.1. In this case, however, we removed any limitation on cache-size for the MOLAP system. We performed this test to simulate the original implementation in [10], where the assumption was made that the entire base table could be cached. We can see from the result depicted in (a), for example, that performance improves across all dimensions uniformly. For the larger data set in (b), the benefit for the MOLAP framework is even more pronounced (relative to the cache-limited version). This is expected, of course. While an unlimited cache is obviously unrealistic for production data warehouses, we note that Figure 4.3 also shows that the ROLAP system still beats the MOLAP model in high dimensions by a ratio of two to one.

Now, in Figure 4.4 we performed a comparative analysis between the cache and the backend DB. In other words, we are looking at the system with and without a native multi-dimensional cache. Data sets in the first test batch are composed of one million tuples and the second batch houses sets of ten million tuples. All sets are skewed with a maximum zipfian value of 1.0. Each batch contains 6 data sets with dimensions ranging from 2 to 7. This evaluation is particularly important as it gives us a measure of real improvement over the current system. In short, it provides analysis that can be used to decide if the new model can be viably included in the current system. As we expected, the Cache works better than the cache-less database

Figure 4.2: Comparative analysis with MOLAP system for (a) number of records = 1 million, zipfian = 0.1 and (b) number of records = 10 million, zipfian = 0.1

Figure 4.3: Comparative analysis with MOLAP system (with no cache-size limit) for (a) number of records = 1 million and (b) number of records = 10 million

across all scenarios, though in lower dimensions the difference between the DB and the Cache is significantly larger. Relatively speaking, the database is less ineffective as dimension count increases, though this is primarily due to the fact that fewer records are being retrieved for queries in this part of the space. Nevertheless, even in seven dimensions, the raw database is roughly three times more expensive than the cache. It should be clear, therefore, that our caching framework can be an important addition to the current Sidera server.

Next, from Figure 4.5 to Figure 4.9, we break down the results from previous experiments, based on the number of queries. Specifically, we use the data set from Figure 4.1 with 10 million records. As one might expect, both caches demonstrate a sub-linear cost increase as query batch size is increased. In other words, the use of the cache allows query resolution times to grow at a slower rate than the increase in query count. That being said, we can also see how the performance curve of the ROLAP cache stabilizes more quickly and more dramatically with an increase in both query count and dimension count. These results are consistent with the previous graphs and underscore the strength of our new model for higher dimensional spaces.

Figure 4.10 to Figure 4.14 examine the caches from a different perspective. Here, we explore the "hit ratio" of our cache system relative to that of the MOLAP system. In other words, how often is the cache being accessed, versus the backend database. We note that, all other things being equal, cache hits rates should be virtually identical since both systems (any cache system, in fact) record the same query history. The results, however, show that the hit ratio improvements for the MOLAP cache system are very minimal, even as query count grows dramatically. Conversely, for our new ROLAP system, the improvement in hit ratio is quite pronounced, more than doubling

Figure 4.4: Comparative analysis with DB

Figure 4.5: Query breakdown for dimension 2

Figure 4.6: Query breakdown for dimension 3

Figure 4.7: Query breakdown for dimension 4

Figure 4.8: Query breakdown for dimension 5

Figure 4.9: Query breakdown for dimension 6

the MOLAP rate in higher dimensions. In short, this is due to the fact that the ROLAP cache system includes a sophisticated update and replacement mechanism that helps to pro-actively identify hotspots in the cache. This results in the database being accessed a lot less *during* the execution of queries.

In Figure 4.15 to Figure 4.19, we ran experiments to further explore the effectiveness of our update mechanism. Specifically, we tested our cache after "turning off" the mechanism that ranked the nodes. In essence, this forces the cache manager to treat all nodes equally and as a consequence, prevents the cache from finding the hotspots. As we can see, the hit ratio drops significantly with the absence of a hotspot detector. This underscores the point made earlier in the thesis. Multi-dimensional caches are simply not effective unless the empty regions in hot areas can be filled in during inactive query periods. Without this feature, even relatively infrequent trips

Figure 4.10: Cache hit ratio for dimension 2

Figure 4.11: Cache hit ratio for dimension 3

Figure 4.12: Cache hit ratio for dimension 4

Figure 4.13: Cache hit ratio for dimension 5

Figure 4.14: Cache hit ratio for dimension 6

to disk will minimize the effectiveness of the cache.

Finally, Figure 4.20 shows a comparison of the time spent to execute 600 queries both with and without the update mechanism. In effect, the performance curves mimic the previous set of results. In this case, we see query resolution times for the hotspot version that are approximately a third of those of the more static cache in higher dimensions.

4.5 Conclusions

In this chapter, we have presented experimental results which demonstrate that our system performs much more effectively than the leading MOLAP based cache system. We have evaluated the cache models with different dimension counts, data sets sizes, and with extreme skew values. In almost all cases, our cache performance is superior

Figure 4.15: Cache hit ratio comparison without update process for dimension 2

Figure 4.16: Cache hit ratio comparison without update process for dimension 3

Figure 4.17: Cache hit ratio comparison without update process for dimension 4

Figure 4.18: Cache hit ratio comparison without update process for dimension 5

Figure 4.19: Cache hit ratio comparison without update process for dimension 6

to that of the MOLAP based cache system. To simulate the environment described in [10], we also examined the impact of eliminating size restrictions on the cache and found that, although this indeed benefits the MOLAP system, our ROLAP framework remains much more effective. Finally, we took a detailed look at both cache hit ratios and query performance with and without the cache framework, and with and without the hotspot detector. Again, significant performance gains were achieved by exploiting the primary features of our new model.

Figure 4.20: Comparative analysis of cache with update and without update

Chapter 5

Conclusions

5.1 Summary

We have described a new relational caching framework that has been integrated into the Sidera parallel OLAP DBMS. The cache — based upon an in-memory variation on the classic R-tree — has shown considerable promise in the low to medium dimensional spaces commonly found in OLAP query environments. In particular, early work has demonstrated the importance of incorporating pro-active insertion policies into cubic caching schemes so as to mitigate the effects of extraneous disk accesses. Furthermore, by exploiting the synergies that naturally exist between the R-tree based disk indexes, the R-tree oriented caching subsystem, and the hierarchy translation facilities that transparently operate on results returned from *either* the cache or the disk, we believe that the current system represents a concrete blueprint for the design of practical high performance OLAP servers.

To summarize the development of R^3-cache, we focus on the following topics.

1. ***Design and development of non-overlapping R-tree***

   We developed the data-structure of our cache based on R-tree modeling. The major difference in our model is that our leaf nodes are non-overlapping. This makes our R-tree more efficient by reducing the query search time and query insertion time into the cache. We developed it by splitting the queries in two different sets. Whenever, we found partial match between two nodes, we split them apart to remove the overlapping part of the node.

2. **Node partition**

   Node partition is required when the size of a leaf node crosses the threshold limit. For our system, this threshold limit is the size of the system page-size. Keeping each individual nodes stored in a single page reduces the memory seek which would have been required otherwise. We developed an efficient node partitioning approach which closely resembles the partitioning approach proposed by Muralikrishna and DeWitt in [3]. According to this approach, nodes are divided into two equal child-nodes each containing exactly the same amount of data. These nodes are partitioned recursively each time along an alternate dimension until the size of the child-nodes are reduced to less or equal to that of the page-size.

3. **Minimizing the node-generation**

   One of the by-products of making the child-nodes non-overlapping was it could potentially generate large number of leaf-nodes. One of the ways, we handled the problem was by merging the adjacent queries. We demonstrated that by merging the adjacent queries, we could actually reduce the number of query in the worst case from $O(n^3)$ to $O(2n)$. We also handled the problem by pre-fetching hotspace area data from the database during the inactive cache time.

4. **Cache management**

   We identified a set of metadata for each of the cache nodes and assigned a "Cache-Value" for each of the nodes using those meta-data. We also identified relationship of metadata that exist between a parent and a child-node. We use the "Cache-Value" of a node to decide on the viability of that cache - whether the cache should be allowed or rejected. We also use this "Cache-Value" to identify the hotspot area among the cache. This feature is especially useful when we pre-fetch data from the data warehouse. Because, we pre-fetch data for the empty spaces within the hotspot area.

5.2 Future work

The research presented in this thesis is the basic framework for developing relational caching model in the data warehouse environment. We identify few areas of future research work which can improve this framework significantly.

1. **Histogram**

   Though our current node-partitioning model(h-tree) performs quite adequately, there is much room for improvement since there has been plenty of research done on different histogram models. Some of these models (mHist, genHist, rK-Hist) [24] are much more improved and they can be easily applied to our caching model.

2. **Hierarchies**

   In our model we use separate trees for separate hierarchies. Each dimension set containing it's own R^3-tree. A new research can look into how R^3-trees in different hierarchies can be connected and useful to each other.

3. **Multi-threading cache update during inactive period**

   Currently, cache gets updated during the inactive period. Though the update process is quite efficient, we can improve it further by making the update process work in a separate thread. This will make sure no incoming queries have to wait for the update process to complete, thus improving query response time.

5.3 Final thoughts

Our objective in this thesis has been to produce a data warehouse cache system which can be seamlessly integrated into the Sidera platform. Since, Sidera was developed on relational OLAP, we set out to develop a cache system based on relational OLAP. We have presented the new R-tree model used for our research purpose. We discussed different cache management techniques used to efficiently manage the cache. Our caching model was tested with data from every spectrum and performed much better than the other comparable caching model. Given the importance of caching system in the data warehouse environment, we believe our research made a meaningful contribution in this area of research.

Bibliography

[1] V. Harinarayan A. Gupta and D. Quass. Aggregate-query processing in data warehousing environments. *In Proceedings of the International Conference on Very Large Databases*, 1995.

[2] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *VLDB '96: Proceedings of the 22th International Conference on Very Large Data Bases*, pages 506–521, 1996.

[3] Jean-Loup Baer and Tien-Fu Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91: Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 176–186, New York, NY, USA, 1991. ACM.

[4] J. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18:509–517, 1975.

[5] K. Beyer and R. Ramakrishnan. Bottom-up computation of sparse and iceberg cubes. In *ACM SIGMOD*, pages 359–370, 1999.

[6] S. Warren C. Hahn and J. London. Edited synoptic cloud reports from ships and land stations over the globe (1982-1991). 2001.

[7] S. Chaudhuri and U. Dayal. An overview of data warehousing and olap technology. *ACM SIGMOD Record*, pages 1:1–20, 1997.

[8] C. Chen and N. Roussopoulos. The implementation and performance evaluation of the adms query optimizer: Integrating query result caching and matching. *In Proceedings of the International Conference on Extending Database Technology*, 1994.

[9] H. Jagadish D. Srivastava, S. Dar and A. Levy. Answering queries with aggragates using views. in proceedings of the international conference on very large databases. *In Proceedings of the International Conference on Extending Database Technology*, 1996.

[10] Prasad Deshpande and Jeffrey F. Naughton. Aggregate aware caching for multidimensional queries. *EDBT '00: Proceedings of the 7th International Conference on Extending Database Technology*, pages 167–182, 2000.

[11] Todd Eavis. Parallel relational olap. *PhD thesis, Dalhousie University, Halifax, NS, Canada*, 2003.

[12] Todd Eavis and Ahmad Taleb. Mapgraph: efficient methods for complex olap hierarchies. In *CIKM '07: Proceedings of the sixteenth ACM conference on Conference on information and knowledge management*, pages 465–474, 2007.

[13] Ernest Teniente Elena Baralis, Stefano Paraboschi. Materialized views selection in a multidimensional database. pages 156–165, 1997.

[14] Edward H. Gornish, Elana D. Granston, and Alexander V. Veidenbaum. Compiler-directed data prefetching in multiprocessors with memory hierarchies. In *ICS '90: Proceedings of the 4th international conference on Supercomputing*, pages 354–368, New York, NY, USA, 1990. ACM.

[15] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *ICDE*, pages 152–159, 1996.

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

[17] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD International Conference on Management of Data*, pages 3–6, 1984.

[18] Venky Harinarayan, Anand Rajaraman, Ullman, and Jeffrey D. Implementing data cubes efficiently. *SIGMOD Rec.*, 25(2):205–216, 1996.

[19] W.H Inmon. Building the data warehouse. *Wiley Comp*, pages 1–3, 1996.

[20] Kamalakar Karlapalem Jian Yang and Qing Li. Algorithms for materialized view design in data warehousing environment. pages 136–145, 1997.

[21] Edward G. Coffman Jr. and Peter J. Denning. *Operating Systems Theory.* Prentice-Hall, 1973.

[22] Ralph Kimball and Joe Caserta. *The Data Warehouse ETL Toolkit: Practical Techniques for Extracting, Cleanin.* John Wiley & Sons, 2004.

[23] Alexander C. Klaiber and Henry M. Levy. An architecture for software-controlled data prefetching. *SIGARCH Comput. Archit. News*, 19(3):43–53, 1991.

[24] John Alexander Lopez. rk-hist: An r-tree based histogram for multi-dimensional selectivity estimation. *Master thesis, Concordia Univeristy, Canada*, 2007.

[25] Todd Mowry and Anoop Gupta. Tolerating latency through software-controlled prefetching in shared-memory multiprocessors. *J. Parallel Distrib. Comput.*, 12(2):87–106, 1991.

[26] M. Muralikrishna and David J. DeWitt. Equi-depth multidimensional histograms. *SIGMOD Rec.*, 17(3):28–36, 1988.

[27] Ralf Schneider Norbert Beckmann, Hans-Peter Kriegel and Bernhard Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD 90: Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331, 1990.

[28] Junho Shim Peter Scheuermann and Radek Vingralek. Watchman: A data warehouse intelligent cache manager. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 51–62, 1996.

[29] Alain Pirotte and Yannis Vassiliou, editors. *VLDB'85, Proceedings of 11th International Conference on Very Large Data Bases, Stockholm, Sweden.* Morgan Kaufmann, 1985.

[30] Allan Kennedy Porterfield. *Software methods for improvement of cache performance on supercomputer applications.* PhD thesis, Houston, TX, USA, 1989. Chairman-K. W. Kennedy.

[31] S. Potamianos S. Chaudhuri, R. Krshnamurthy and K. Shim. Optimizaing queries with materialized views. *In Proceedings of International Conference on Data Engineering*, 1985.

[32] Junho Shim, Peter Scheuermann, and Radek Vingralek. Dynamic caching of query results for decision support systems. *SSDBM '99: Proceedings of the 11th International Conference on Scientific on Scientific and Statistical Database Management*, page 254, 1999.

[33] Jagadish H.V. Srivastava D., Dar S. and Levy A.Y. Answering queries with aggregation using views. *Proceedings of 22th International Conference on Very Large Data Bases (VLDB96, Bombay, India, Sept. 3-6)*, pages 318–329, 1996.

[34] Michael Stonebraker. Virtual memory transaction management. *Operating Systems Review*, 18(2):8–16, 1984.

[35] Ahmad Taleb. A framework for the manipulation of olap hierarchies. *Computer*, 2007.

[36] A. Rajaraman V. Harinarayan and J. Ullman. Implementing data cubes. *Proceedings of the 1996 ACM SIGMOD Conference*, 28(1):2–5, 1996.

[37] George K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison-Wesley (Reading MA), 1949.