EFFICIENT INDEXING TECHNIQUES ON DATA WAREHOUSE

Bhosale P., Bidkar N, Hirave T., Kanase P., Magar M.

Abstract— Recently, data warehouse system is becoming more and more important for decision-makers. Most of the queries against a large data warehouse are complex and iterative. The ability to answer these queries efficiently is a critical issue in the data warehouse environment. If the right index structures are built on columns, the performance of queries, especially ad hoc queries will be greatly enhanced. In this paper, we provide an evaluation of indexing techniques being studied/used in both academic research and industrial applications.

Index Terms— Data warehouse, Indexing, Information Storage and Retrieval, P+ tree, R* tree, OLTP, Database.

1 INTRODUCTION

A Data warehouse (DW) is a large repository of information accessed through an Online Analytical Processing (OLAP) application. This application provides users with tools to iteratively query the DW in order to make better and faster decisions. The information stored in a DW is clean, static, integrated, and time varying, and is obtained through many different sources. Such sources might include Online Transaction Processing (OLTP) or previous legacy operational systems over a long period of time. Requests for information from a DW are usually complex and iterative queries. Complex queries could take several hours or days to process because the queries have to process through a large amount of data.

In the last 25 years many indexing techniques have been proposed for the efficient storage and retrieval of multidimensional data. For the one-dimensional case, the ubiquitous B+ tree has been incorporated in all commercial and open source database management systems. Many more sophisticated data structures have been proposed to handle the problem of manipulating in an efficient manner enormous sizes of multidimensional data. Most of them try to solve problems concerning range queries and k nearest neighbour (kNN) queries. Difficulties arise in higher dimensions where the problem of the so called "dimensionality curse" has the effect that the higher the dimension in question the more these index structures behave like or even worse than the sequential scan in solving problems like similarity search queries.

These indexing methods usually take advantage of many factors like the manner that space is occupied by the data in question or some characteristics of the way that data space is decomposed that can lead to translating the multidimensional problem into a single-dimensional one that can be efficiently handled by a B+tree. First, we survey some of the most notable indexing structures that have been proposed in the literature, especially the R*tree (successor of R-tree), the Hybrid tree , the P+tree and the iDistance , and then we try to study and investigate through experimentation various factors that influence these indexes when used to solve kNN queries. These factors are the data dimensionality and the size of the indexed data that usually arise in real world datasets.

2 PURPOSE OF WORK

Requests for information from a DW are usually complex and iterative queries. Such complex queries could take several hours or days to process because the queries have to process through a large amount of data. A majority of requests for information from a data warehouse involve dynamic ad hoc queries. Users can ask any question at any time for any reason against the base table in a data warehouse. The ability to answer these queries quickly is a critical issue in the data warehouse environment.

Among the various solutions such as summary tables, indexes, parallel machines, etc. to speed up query processing, Indexing is the best key to overcome this problem.

3 TECHNOLOGIES AND METHODOLOGY

3.1 DATABASE

The term or expression of database originated within the computer industry. A possible definition is that a database is a structured collection of records or data which is stored in a computer so that a program can consult it to answer queries. The records retrieved in answer to queries become information that can be used to make decisions. The computer program used to manage and query a database is known as a Database Management System (DBMS). The central concept of a database is that of a collection of records, or pieces of knowledge. Topically, for a given database, there is a structural description of the type of facts held in that database: this description is known as a schema. The schema describes the objects that are represented in the database, and the relationships among them. There are a number of different ways of organizing a

schema, that is, of modelling the database structure: these are known as database models (or data models). The model in most common use today is the relational model that represents all information in the form of multiple related tables each consisting of rows and columns. This model represents relationships by the use of values common to more than one table. The term *database* refers to the collection of related records, and the software should be referred to as the *database management system* or DBMS. When the context is unambiguous, however, many database administrators and programmers use the term *database* to cover both meanings. Database management systems are usually categorized according to the data model that they support: relational, object-relational, network, and so on. The data model will tend to determine the query languages that are available to access the database. A great deal of the internal engineering of a DBMS, however, is independent of the data model, and is concerned with managing factors such as performance, concurrency, integrity, and recovery from hardware failures.

3.2 DATA WAREHOUSE

Data warehousing is a collection of decision support technologies, aimed at enabling the knowledge worker (executive, manager, and analyst) to make better and faster decisions. A data warehouse is a "subject-oriented, integrated, time varying, non-volatile collection of data that is used primarily in organizational decision making."1 Typically, the data warehouse is maintained separately from the organization's operational databases. There are many reasons for doing this. The data warehouse supports on-line analytical processing (OLAP), the functional and performance requirements of which are quite different from those of the on-line transaction processing (OLTP) applications traditionally supported by the operational databases.

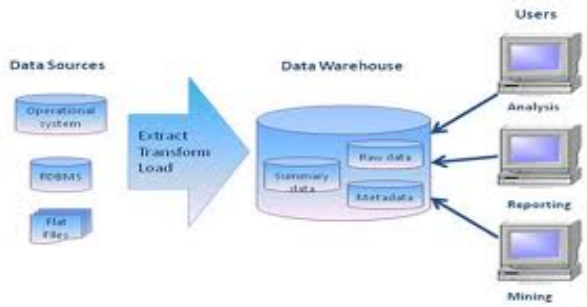

Fig. Data warehouse

Figure 1 shows a typical data warehousing architecture. It includes tools for extracting data from multiple operational databases and external sources; for cleaning, transforming and integrating this data; for loading data into the data warehouse; and for periodically refreshing the warehouse to reflect updates at the sources and to purge data from the warehouse, perhaps onto slower archival storage. In addition to the main warehouse, there may be several departmental data marts. Data in the warehouse and data marts is stored and managed by one or more warehouse servers, which present multidimensional views of data to a variety of front end tools: query tools, report writers, analysis tools, and data mining tools. Finally, there is a repository for storing and managing metadata, and tools for monitoring and administering the warehousing system. The warehouse may be distributed for load balancing, scalability, and higher availability. In such a distributed architecture, the metadata repository is usually replicated with each fragment of the warehouse, and the entire warehouse is administered centrally. An alternative architecture, implemented for expediency when it may be too expensive to construct a single logically integrated enterprise warehouse, is a federation of warehouses or data marts, each with its own repository and decentralized administration. Designing and rolling out a data warehouse is a complex process, consisting of the following activities.

- Define the architecture, do capacity planning, and select the storage servers, database and OLAP servers, and tools.
- Integrate the servers, storage, and client tools.
- Design the warehouse schema and views.
- Define the physical warehouse organization, data placement, partitioning, and access methods.
- Connect the sources using gateways, ODBC drivers, or other wrappers.
- Design and implement scripts for data extraction, cleaning, transformation, load, and refresh.
- Populate the repository with the schema and view definitions, scripts, and other metadata.
- Design and implement end-user applications.
- Roll out the warehouse and applications.

3.3 INDEXING TECHNIQUES

Recently, data warehouse system is becoming more and more important for decision-makers. Most of the queries against a large data warehouse are complex and iterative. The ability to answer these queries efficiently is a critical issue in the data warehouse environment. If the right index structures are built on columns, the performance of queries, especially ad hoc queries will be greatly enhanced. In this project, we provide an evaluation of indexing techniques being studied/used in both academic research and industrial applications. In addition, we identify the factors that need to be considered when one wants to build a proper index on base data.

There are many solutions to speed up query processing such as summary tables, indexes, parallel machines, etc. However when an unpredicted query arises, the system must scan, fetch, and sort the actual data, resulting in performance degradation. Whenever the base table changes, the summary tables have to be recomputed. Also building summary tables often supports only known frequent queries, and requires more time and more space than the original data. Because we cannot build all possible summary tables, choosing which ones to be built is a difficult job. Moreover, summarized data hide valuable information. For example, we cannot know the effectiveness of the promotion on Monday by querying weekly summary. Indexing is the key to achieve this objective without adding additional hardware.

We attempt a fair comparison of many state of the art indexing structures designed exclusively to index multi-dimensional points like the Hybrid tree, B-tree, Projection index, Bitmap index, Pure Bitmap index, iDistance and the P+tree. We include in our comparison the R*tree, a state of the art index designed both for multidimensional points and regions. It is an improvement of the well-known R-tree, and also has been revised and improved further recently[1]. There are some indexing techniques. Some of them are as follows:

B-Tree Index

The B-Tree Index is the default index for most relational database systems. The top most level of the index is called the root. The lowest level is called the leaf node. All other levels in between are called branches. Both the root and branch contain entries that point to the next level in the index. Leaf nodes consisting of the index key and pointers pointing to the physical location (i.e., row ids) in which the corresponding records are stored.

Bitmap Index

The bitmap representation is an alternate method of the row ids representation. It is simple to represent, and uses less space- and CPU-efficient than row ids when the number of distinct values of the indexed column is low. The indexes improve complex query performance by applying low-cost Boolean operations such as OR, AND, and NOT in the selection predicate on multiple indexes at one time to reduce search space before going to the primary source data.

Pure Bitmap Index

Pure Bitmap Index was first introduced and implemented in the Model 204 DBMS. It consists of a collect of bitmap vectors each of which is created to represent each distinct value of the indexed column. A bit i in a bitmap vector, representing value x, is set to 1 if the record i in the indexed table contains x. To answer a query, the bitmap vectors of the values specified in the predicate condition are read into memory. If there are more than one bitmap vectors read, a Boolean operation will be performed on them before accessing data. Most of commercial data warehouse products (e.g., Oracle, Sybase, Informix, Red Brick, etc.) implement the Pure Bitmap Index.

Join Index

A Join Index is built by translating restrictions on the column value of a dimension table (i.e., the gender column) to restrictions on a large fact table. The index is implemented using one of the two representations: row id or bitmap, depending on the cardinality of the indexed column.

P –tree

An entry in an internal node of the P-tree has the form <K,Pr>, where K is an *entry key* and Pr is the pointer to a child of the node. An entry key K of length $l(\geq 1)$ has the format of #i.c1.c2……cl, where #i is the id of a level-0 list in the list database and ci(≥1) is the *position offset* of a level-i list, explained below. T(Pr) denotes the subtree under branch Pr. Entry keys can have different length l so that nesting depth of lists can grow and shrink dynamically anywhere in a list.

4 Research Methodology

4.1 P+ tree

The basic idea of the P+-tree is to divide the space into subspaces and then apply the Pyramid technique in each subspace. To realize this, we first divide the space into clusters which are essentially hyper rectangles. We then transform each subspace into a hypercube so that we can apply the Pyramid technique on it. At the same time, the transformation makes the top of the pyramids located at the cluster center. Assuming that real queries follow the same distribution as data, most of the queries would be located around the top of the pyramids, that is, the "good position". Even if some queries may be located at the corner or edge of the cluster and therefore causes a large region to be accessed, the data points accessed are not prohibitively large because most of the data points are gathered at the cluster center. In addition, the region accessed by a query is significantly reduced by space division. Thus, the P+-tree can alleviate the inefficiencies of the Pyramid technique. We note that although we cluster the space into subspaces,

our scheme also works for uniform data since uniform data is a special case of clustered data. While uniform data does not benefit from the transformation, dividing the space into subspaces is still an effective mechanism for performance improvement. To facilitate building the P+-tree and query processing, we need an auxiliary structure called the *space-tree*, which is built during the space division process. The leaf nodes of the space-tree store information about the transformation. We will first introduce the data transformation, so that readers know what information is stored. Then, we present the space division process. At last, we show how the P+-tree is constructed.

4.2 R* TREE

R*-trees are a variant of R-trees used for indexing spatial information. R*-trees support point and spatial data at the same time with a slightly higher cost than other R-trees. It was proposed by Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger in 1990.

The R*-tree is a data partitioning structure that indexes MBRs (minimum bounding rectangles). The minimization of both coverage and overlap of the MBRs influences the performance of R* tree. When overlap occurs on data query or insertion, more than one branch of the tree needs to be expanded and traversed (due to storage redundancy). When the coverage is minimized this has the effect of improving pruning performance, so whole pages are excluded from search more often.

The R*-tree attempts to reduce both, with a combination of a revised node split algorithm and the concept of forced reinsertion when nodes overflow. This is based on the observation that R-tree structures are highly sensitive to the order in which their entries are inserted, so an insertion-built (rather than bulk-loaded) structure is likely to be sub-optimal. So the deletion and reinsertion of some entries allows them to "find" a place in the tree that may be more appropriate than their original location. When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. This produces better-clustered groups of entries in nodes, with the effect that node coverage is reduced. Furthermore, actual node splits are often postponed, causing average node occupancy to become higher. Re-insertion can be seen as a method of incremental tree optimization triggered on node overflow.

R*-Tree built by repeated insertion (in ELKI). There is little overlap in this tree, resulting in good query performance. Red and blue MBRs are index pages, green MBRs are leaf nodes. Minimization of both coverage and overlap is crucial to the performance of R-trees. Overlap means that, on data query or insertion, more than one branch of the tree needs to be expanded (due to the way data is being split in regions which may overlap). A minimized coverage improves pruning performance, allowing to exclude whole pages from search more often, in particular for negative range queries.

The R*-tree attempts to reduce both, using a combination of a revised node split algorithm and the concept of forced reinsertion at node overflow. This is based on the observation that R-tree structures are highly susceptible to the order in which their entries are inserted, so an insertion-built (rather than bulk-loaded) structure is likely to be sub-optimal. Deletion and reinsertion of entries allows them to "find" a place in the tree that may be more appropriate than their original location.

When a node overflows, a portion of its entries are removed from the node and reinserted into the tree. (In order to avoid an indefinite cascade of reinsertions caused by subsequent node overflow, the reinsertion routine may be called only once in each level of the tree when inserting any one new entry.) This has the effect of producing more well-clustered groups of entries in nodes, reducing node coverage. Furthermore, actual node splits are often postponed, causing average node occupancy to rise. Re-insertion can be seen as a method of incremental tree optimization triggered on node overflow.

5 CONCLUSION

In this paper we have proposed various indexing techniques. Some of which are more efficient for data warehouse such as P+ and R* tree, which are briefly describe in this paper for the purpose of further implementation.

REFERENCES

[1] Rui Zhang Beng Chin Ooi Kian-Lee Tan "Making the Pyramid Technique Robust to Query Types and Workloads" Department of Computer Science National University of Singapore, Singapore 117543

[2] Nikolaos Kouiroukidis, Georgios Evangelidis "Efficient indexing methods in the data mining context" *Department of Applied Informatics, University of Macedonia 156 Egnatia Str., Thessaloniki, 54006, Greece*

[3] Sirirut Vanichayobon, Le Gruenwald "Indexing Techniques for Data Warehouses' Queries" School of Computer Science Norman, OK, 73019

[4] Surajit Chaudhuri Microsoft Research, Redmond, Umeshwar Dayal Hewlett-Packard Labs, Palo Alto "An Overview of Data Warehousing and OLAP Technology"