Indexing Techniques for Data Warehouses' Queries

Sirirut Vanichayobon
Le Gruenwald
The University of Oklahoma
School of Computer Science
Norman, OK, 73019
sirirut@cs.ou.edu
gruenwal@cs.ou.edu

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. In addition, we identify the factors that need to be considered when one wants to build a proper index on base data.

1. Introduction

A data warehouse (DW) is a large repository of information accessed through an Online Analytical Processing (OLAP) application [CD97]. 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 [Inm93]. 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 of what happened in a business such as "Finding the products' types, units sold and total cost that were sold last week for all stores in west region?". Most of the queries contain a lot of join operations involving a large number of records. Also, aggregate functions such as *group-by* are very common in these 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 ([TPC98], [APB98]); 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.

There are many solutions to speed up query processing such as summary tables, indexes, parallel machines, etc. The performance when using summary tables for predetermined queries is good. 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.

The objectives of this paper are to identify factors that need to be considered in order to select a proper indexing technique for data warehouse applications, and to evaluate indexing techniques being studied/used in both academic research and industrial applications. The rest of the paper is organized as follows. In Section 2 we discuss the important issues that we have to consider when building/selecting

an indexing technique for the DW. In Section 3 we evaluate existing indexing techniques currently used in data warehouses. In Section 4 we give conclusions and present directions for future work.

2. Indexing Issues

Indexes are database objects associated with database tables and created to speed up access to data within the tables. Indexing techniques have already been in existence for decades for the OLTP relational database system but they cannot handle large volume of data and complex and iterative queries that are common in OLAP applications. The differences between the OLAP and the OLTP applications, shown in Table 1, determine that some new or modified techniques have to be implemented since the existing indexing techniques are inadequate for OLAP applications.

In the following subsections we discuss the important issues that we have to consider in order to design/select the right index structure to support DW's queries [Col96].

OLTP	OLAP
♦ Current data	♦ Current data as well as history.
♦Used to support transaction processing	♦ Used to support the business interests
♦ Clerical data processing tasks	♦ Decision support tasks
♦ Simple and known queries	♦ Ad hoc, complex, and iterative queries which access millions records and perform a lot of joins and aggregates
♦ A few tables involved and unlikely to be scanned	♦ Multiple tables involved and likely to be scanned
♦ Small foundset	♦ Large foundset
♦ Short transactions	♦ Long transactions
♦ Update/Select	♦ Select (Read only)
♦ Real time update	♦ Batch update
♦ Unique index	♦ Multiple index
♦ Known access path	♦ Do not know access path until users start asking queries
♦ Detail row retrieval	♦ Aggregation and group by
♦ High selectivity queries	♦ Low selectivity queries
♦ Low I/O and processing	♦ High I/O and processing
♦ Response time does not depend on database size	♦ Response time depends on database size
♦ Data model: entity relational	♦ Data model: multidimensional

Table 1: Summarizes the main differences between OLTP and OLAP systems.

2.1 Factors used to determine which indexing technique should be built on a Column

a) Characteristics of indexed column

A column has its own characteristics which we can use to choose a proper index. These characteristics are given below:

- *Cardinality data*: The cardinality data of a column is the number of distinct values in the column. It is better to know that the cardinality of an indexed column is low or high since an indexing technique may work efficiently only with either low cardinality or high cardinality.
- *Distribution*: The distribution of a column is the occurrence frequency of each distinct value of the column. The column distribution guides us to determine which index type we should take.
- *Value range*: The range of values of and indexed column guides us to select an appropriate index type. For example, if the range of a high cardinality column is small, an indexing technique based on bitmap should be used. Without knowing this information, we might use a B-Tree resulting in a degradation of system performance.

b) Understanding the data and the usage in the SQL language

Knowing the columns that will be queried helps us choose appropriate index types for them. For example, which columns will likely be a part of the selection list, join constraints, application constraints, the ORDER BY clause, or the GROUP BY clause?

2.2 Developing a new indexing technique for data warehouse's queries.

The following are the characteristics that we have to concern with when developing a new indexing technique:

a) The index should be small and utilize space efficiently.

b) The index should be able to operate with other indexes to filter out the records before accessing raw data.

c) The index should support ad hoc and complex queries and speed up join operations.

d) The index should be easy to build (easily dynamically generate), implement and maintain.

3. Evaluation of Existing Indexing Techniques in Data Warehouses

In data warehouse systems, there are many indexing techniques. Each existing indexing technique is suitable for a particular situation. In this section we describe several indexing techniques being studied/used in both academic research and industrial applications. We will use the example in Figure 1 to explain the indexing techniques throughout the paper. The figure illustrates an example of a star

schema with a central fact table called SALE and two dimension tables called PRODUCT and CUSTOMER.

3.1 The B-Tree Index

The B-Tree Index is the default index for most relational database systems [KRRT98]. 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. A B-Tree Index for package_type of the PRODUCT table is shown in Figure 2.

PRODUCT TABLE

Product ID	Weight	Size	Package_Type
P10	10	10	A
P11	50	10	B
P12	50	10	A
P13	50	10	C
P14	30	10	A
P15	50	10	B
P16	50	10	D
P17	5	10	H
P18	50	10	I
P19	50	10	E
P21	40	10	I
P22	50	10	F
P23	50	10	J
P24	50	10	G
P25	10	10	F
P26	50	10	F
P27	50	10	J
P28	20	10	H
P29	50	10	G
P30	53	10	D

CUSTOMER TABLE

Customer_ID	Gender	City	State
C101	F	Norman	OK
C102	F	Norman	OK
C103	M	OKC	OK
C104	M	Norman	OK
C105	F	Ronoake	VA
C106	F	OKC	OK
C107	M	Norman	OK
C108	F	Dallas	TX
C109	M	Norman	OK
C110	F	Moore	OK

SALE TABLE

Product_ID	Customer_ID	Total_Sale
P10	C105	100
P11	C102	100
P15	C105	500
P10	C107	10
P10	C106	100
P10	C101	900
P11	C105	100
P10	C109	20
P11	C109	100
P10	C102	400
P13	C105	100

Figure 1: An example of the PRODUCT, CUSTOMER and SALE table.

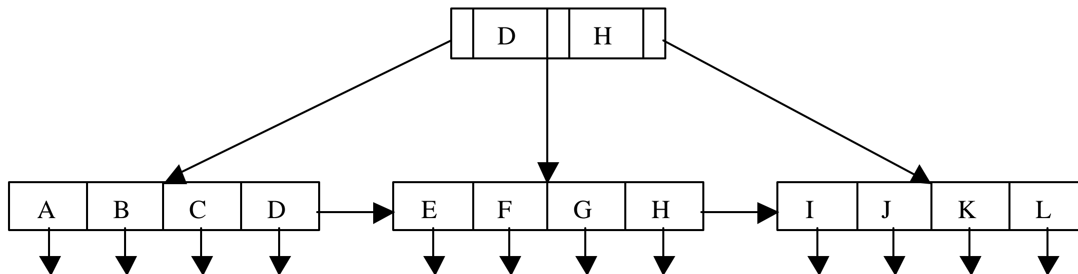

Figure 2: The B-Tree Index on the package_type column of the PRODUCT table.

The B-Tree Index is popular in data warehouse applications for high cardinality column such as names since the space usage of the index is independent of the column cardinality. However, the B-Tree Index has characteristics that make them a poor choice for DW's queries. First of all, a B-Tree index is of no value for low cardinality data such as the gender column since it reduces very few numbers of I/Os and may use more space than the raw indexed column. Secondly, each B-Tree Index is independent and thus cannot operate with each other on an index level before going to the primary source. Finally, the B-Tree Index fetches the result data ordered by the key values which have unordered row ids, so more I/O operations and page faults are generated.

3.2 Projection Index [OQ97]

A Projection Index on an indexed column ***A*** in a table ***T*** stores all values of ***A*** in the same order as they appear in ***T***. Each row of the Projection Index stores one value of ***A***. The row order of value ***x*** in the index is the same as the row order of value ***x*** in ***T*** [OQ97]. Figure 3 shows the Projection Index on package_type of the PRODUCT table. Normally, the queries against a data warehouse retrieve only a few of the table's columns; so having the Projection Index on these columns reduces tremendously the cost of querying because a single I/O operation may bring more values into memory. Sybase builds a Projection Index under the name of *FastProjection Index* on every column of a table.

3.3 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. Many variations of the Bitmap Index (Pure Bitmap Index, Encoded Bitmap, etc.) have been introduced, aiming to reduce space requirement as well as improve query performance.

a) Pure Bitmap Index [O'N87]: 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***. Figure 3 shows an example of the Pure Bitmap Index on the package_type column of the PRODUCT table. The Pure Bitmap Index on this column is the collection

of 12 bitmap vectors, says {B_A, B_B, B_C, B_D, B_E, B_F, B_G, B_H, B_I, B_J, B_K, and B_L }, one for each package type. 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. However, the sparsity problem occurs if the Pure Index is built on high cardinality column which then requires more space and query processing time to build and answer a query. Most of commercial data warehouse products (e.g., Oracle, Sybase, Informix, Red Brick, etc.) implement the Pure Bitmap Index.

Package_Type	B_A	B_B	B_C	B_B	B_E	B_F	B_G	B_H	B_I	B_J	B_K	B_L
A	1	0	0	0	0	0	0	0	0	0	0	0
B	0	1	0	0	0	0	0	0	0	0	0	0
L	0	0	0	0	0	0	0	0	0	0	0	1
C	0	0	1	0	0	0	0	0	0	0	0	0
:	:	:	:	:	:	:	:	:	:	:	:	:
G	0	0	0	0	0	0	1	0	0	0	0	0
D	0	0	0	1	0	0	0	0	0	0	0	0

(a) Projection **(b) Pure Bitmap Index**

Figure 3: An example of the Projection Index and Pure Bitmap Index on the package_type column of PRODUCT table.

Encoded Bitmap Index [WB97]: An Encoded Bitmap Index on a column ***A*** of a table ***T*** consists of a set of bitmap vectors, a lookup table, and a set of retrieval Boolean functions. Each distinct value of a column A is encoded using a number of bits each of which is stored in a bitmap vector. The lookup table stores the mapping between A and its encoded representation. IBM implements this index in DB2. Comparing with the Pure Bitmap Index, the Encoded Bitmap Index improves the space utilization, and solves sparsity problems. The size of the Encoded Bitmap Index built on the high cardinality column is less than the Pure Bitmap Index. Having a well defined encoding scheme, a Boolean operation can perform on the retrieval functions before retrieving the data, and lead to a reduction of the number of bitmap vectors read. Its performance is degraded with equality queries since we have to search all the bitmap vectors. The index needs to be rebuilt if we run out of bits to represent new values.

3.3 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 [Vald87] or bitmap [OG95], depending on the cardinality of the indexed column. A bitmap representation, which is called Bitmap Join Index, is used with the low cardinality

data while a row id representation is used with a high cardinality. In DW, there are many join operations involved; so building Join Indexes on the joining columns improves query-processing time. For example, Bitmap Join Indexes on the gender column in the SALE table can be built by using the gender column in the CUSTOMER table and the foreign key customer id in the SALES table. Note that the Sales table does not contain the gender column. The Bitmap Join Index for gender equal to male is created by setting a bit corresponding to a row for customer_id whose gender is 'M' to 1 in the Sales Table. Otherwise, the bit is set to 0 as shown in Figure 4.

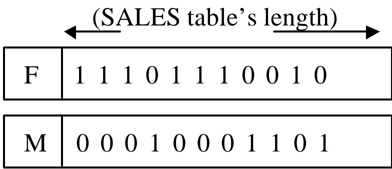

Figure 4: An example of a Bitmap Join Index on column gender in the SALE table.

If a bitmap vector is built by translating restrictions on the column values from several joined tables at once (e.g. gender and product type in the different dimension tables) then it is called a Multiple Bitmap Join Index.

3.4 Summary of Evaluation of Existing Indexing Techniques

Table 2 summarizes the key features of the evaluated indexed techniques and also include the commercial data warehouse products that implement these techniques.

Indexing Techniques	Characteristics	Advantages	Disadvantages	Implementing Commercial Systems
B-Tree Index	Two representations (row id and bitmap) are implemented at the leaves of the index depending on the cardinality of the data.	• It speeds up known queries. • It is well suited for high cardinality. • The space requirement is independent of the cardinality of the indexed column. • It is relatively inexpensive when we update the indexed column since individual rows are locked.	• It performs inefficiently with low cardinality data • It does not support ad hoc queries. More I/O operations are needed for a wide range of queries. • The indexes can not be combined before fetching the data.	• Most of commercial products (Oracle, Informix, Red Brick
Pure Bitmap Index	An array of bits is utilized to represent each unique column value of each row in a table, setting the bits corresponding to the row either *ON*(valued 1) or *OFF*(valued 0). The equality encoding scheme is used.	• It is well suited for low-cardinality columns. • It utilizes bitwise operations. • The indexes can be combined before fetching raw data. • It uses low space • It works well with parallel machine. • It is easy to build. • It performs efficiently with columns involving scalar functions (e.g., COUNT). • It is easy to add new indexed value. • It is suitable for OLAP.	• It performs inefficiently with high cardinality data. • It is very expensive when we update index column. The whole bitmap segment of the updated row is locked so the other row can not be updated until the lock is released. • It does not handle spare data well.	• Oracle • Informix • Sybase • Informix • Red Brick • DB2
Encoded Bitmap Index	The index is the binary Bit-Sliced Index built on the attribute domain	• It uses space efficiently. • It performs efficiently with wide range query.	• It performs inefficiently with equality queries. • It is very difficult to find a good encoding scheme. • It is rebuilt every time when a new indexed value for which we run out of bit to represent is added.	• DB2
Bitmap Join Index	The index is built by restriction of a column on the dimension table in the fact table.	• It is flexible. • It performs efficiently. • It supports star queries.	• The order of indexed column is important.	• Oracle • Informix • Red Brick
Projection Index	The index is built by storing actual values of column(s) of indexed table.	• It speeds up the performance when a few columns in the table are retrieved.	• It can be used only to retrieve raw data (i.e., column list in selection).	• Sybase

Table 2: Existing Indexing Technique.

4. Conclusions and Future Work

The ability to extract data to answer complex, iterative, and ad hoc queries quickly is a critical issue for data warehouse applications. A proper indexing technique is crucial to avoid I/O intensive table scans against large data warehouse tables. The challenge is to find an appropriate index type that would improve the queries' performance. B-Tree Indexes should only be used for high cardinality data and predicted queries. Bitmap Indexes play a key role in answering data warehouse's queries because they have an ability to perform operations on index level before retrieving base data. This speeds up query processing tremendously. Variants of Bitmap Indexes have been introduced to reduce storage requirement and speed up performance. Recently, most commercial data warehouse products except Teradata database implement Bitmap Indexes. Finding a new indexing technique based on Bitmap Indexes is still the interesting research area. To further speed up queries processing, after using Bitmap Indexes to evaluate query predicates, Projection Indexes can be used to retrieve the columns that satisfy the predicates. However, good index structures are useless if we do not employ an intelligent query optimizer to select a suitable indexing technique to process queries. Data mining techniques could be used to develop an intelligent optimizer. Paralleling is another issue that we should consider.

References

[APB98] OLAP Council, "APB-1 OLAP Benchmark Release II", November 1998. http://www.olapcouncil.org.

[CD97] S. Chaudhuri and U. Dayal, "An Overview of Data Warehousing and OLAP Technology", ACM SIGMOD RECORD, 26(1):65-74, March 1997.

[Col96] G. Colliat, OLAP, "Relational and Multimedimensional Database System", SIGMOD Record, 25(3):64-69, Sept. 1996

[EN94] R. Elmasri, and S.B. Navathe, "Fundamentals of Database Systems", 2nd Edition, Addison-Wisley Publishing Company, 1994.

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman, "Implementing Data Cubes Efficiently", In Proc. of the ACM SIGMOD Conf. on Management of Data, Jun. 1996

[Inm93] W.H. Inmon, "Building the Data Warehouse", John Wiley & Sons, 1993.

[KRRT98] R. Kimball, L. Reeves, M. Ross and W. Thornthwaite, "The Data Warehouse Lifecycle Toolkit : Expert Methods for Designing, Developing, and Deploying Data Warehouses", John Wiley & Sons, Aug. 1998

[OG95] P. O'Neil and G. Graefe, "Multi-Table joins through Bitmapped join indices", SIGMOD Record, Vol. 24, No. 3, Sep. 1995

[O'N87] P. O'Neil, "Model 204 Architecture and Performance", Springer-Verlag Lecture Notes in Compuer Science 359, 2nd Intl. Workshop on High Performance Transactions Systems, Asilomar, CA, Sept 1987

[OQ97] P. O'Neil and D. Quass, "Improved Query Performance with Variant Indexes",SIGMOD,1997

[TPC98] Transaction Processing Performance Council (TPC), "TPC Benchmark D, Decision Support", Standard Specification Revision 2.0.1, December 5, 1998, http://www.tpc.org.

[WB97] MC. Wu and A. Buchmann, "Encoded Bitmap Indexing for Data Warehouses", DVS1, Computer Science Department,Technische University, 1997