Bulkloading of Bitmap Indices

Naveen Prakash
CS764: Topics in Database Management Systems
naveen@cs.wisc.edu

Abstract:

The conventional indexing technique designed for the traditional database systems does not use the read-only characteristics of the data warehouse environment. Recently, Bitmap Indexing has been proposed as a candidate indexing structure suitable for the read-only environments, like those of decision support systems. The current paper presents a short literature survey of the various clone of this indexing structure, before presenting the design which is being implemented in Paradise. We present algorithms that can be used for query evaluation using bitmaps. Wherever possible a comparative study of bitmaps and the traditional RID list representation has been provided, with space-time tradeoffs involved.

1 Introduction

As the datasize increase, the hunt for search mechanism that scale up to the increase in datasize continue. Data warehousing has emerged as a convenient method to store data for later analysis. Data warehouses contains data for a large number of independent sources. The queries involved are mostly unpredictable and includes filtering, aggregating and grouping the data in many different ways.

Consider the star-join schema shown in figure 1. It consists of a central fact table Sales which contain an entry for each sale. The various dimensions include time, location, product, customer and promotion. The size of fact table is huge for chains like Wallmart. Queries over the fact table includes join with its various dimensions. One such query could be the amount of sale for a particular product in a particular state during some part of the year. Such queries help in preparing business strategies for better servicing the customer, and one requires to get results as quickly as possible. There can be many methods to speed up such queries. They can be broadly categorized as hardware solutions and software solutions.

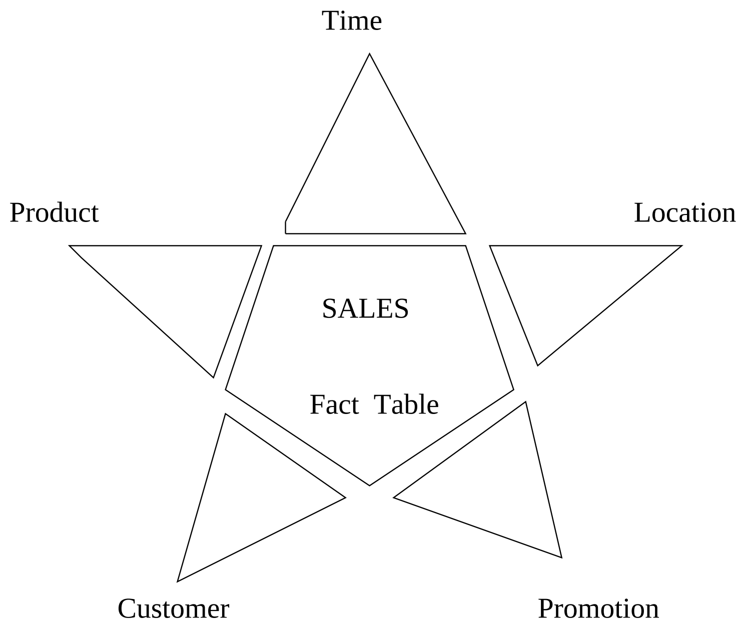
We can have faster CPU's, but the size of fact table has grown at much faster rate than the available MIPS. Further more, with the increase in the data size, the disk IO becomes a bottleneck. One can go for parallel database, but this alone cannot provide a satisfactory solution. Most of these queries require to retrieve matching tuples and do aggregation on it. Indexing can be used to reduce the number of IO's required, but the traditional indexing technique like B-Tree does not provide for efficient queries over more than one dimension. The Software solution to this problem include fancy indexing technique like inverted file system, bitmaps and specialized multidimensional databases. An easy way out is to build countless indexes and summary table to prevent scanning the whole fact table. This would, no doubt, lead to degradation in performance both on the time scale and disk space. DISC came out with a database indexing technology called OMNIDEX[1], which is like a search engine that resides between the existing database and the application. It creates indexes on the specified columns of the table at the installation time at very high speed and stores them in compressed format. Most queries are run using its index-only-scan and only RIDs are used to retrieve the records. RIDs are the most efficient way of getting a record. For detailed discussion of different indexing scheme, in particular Inverted File Indexing, for Data warehouses one can refer to [2].

In this paper we present the issues associated with Bitmap Indexing which were first used in MODEL 204. Now various DBMS vendors like Oracle and Sybase have started providing it as one of the indexing structure.

2 B-tree with bitmaps

B-Tree has been used for providing an efficient access path in almost all the traditional database. It adapts dynamically to the changes in databases. It stores key-RID pair in its leaf pages. The B-Tree index is used to retrieve tuples satisfying the primary term in the where clause, and then all these tuples are checked for the secondary term on fly. The other option could be to use different indexes to retrieve tuples satisfying different term in the where clause and then take an intersection of it. Usually the selectivity of all terms is much lower than the selectivity of each individual terms in data warehouse environment. This will lead to retrieval of large number of tuples in the first case. For the second case, a large number of B-tree index on different combination of attributes will be required. As the queries in such environment are mostly unpredictable, one may require to build 2**n -1 indexes on a relation with n attributes. Furthermore two B-Tree on two

Figure 1: Star Schema Example



Star Join Schema of SALES, Time, Location, Promotion, Customer, and Product

SALES(sid, day, lid, pid, cid, bid, unit sales, dollar sales,...)

Time(day,week,month,year,...)
Location(lid,street,city,state,...)
Promotion(pid,....)
Customer(cid,....)
Product(bid,brand,...)

different relation can not be easily combined.

We will use the B-Tree structure, but instead of storing RID list with each distinct key value, we will store a bitmap corresponding to it. Conceptually bitmaps are a very simple way to store data. For every distinct value for a attribute X, we store a corresponding bitmap. The number of such bitmaps will therefore be equal to size of the domain of that attribute. The size of bitmaps will be equal to the cardinality of the relation.

For example, if one of the attribute in a relation is gender, then there will be two bitmaps, corresponding to Male and Female. If the nth tuple has Male as its value, the the nth bit of the Male bitmap will be set to one, while that of Female bitmap set to zero. In the Btree, one would have stored the RIDs of tuple with Male as its attribute value with the Male key and similarly for the Female key. As there is an implicit ordering of records, its very costly to update bitmap index.

2.1 Space Requirements

Bitmaps can only be used for domains with discrete values. Also its not wise to use them on attributes which can take large number of values, like the name attribute in a student record. The space requirement for RID list does not depend on the size of the attribute domain. It depends only on the number of tuples in a relation. For bitmaps, the space requirements shoots as the number of distinct value for the attribute (on the index is being built) increases. It will be very costly to build bitmap index on a key. This means a bitmap with only one bit set is stored with every distinct key. Number of such bitmaps will be equal to the cardinality of the relation. On the other extreme, if the number of distinct attribute value is two (as in the first example), then we require to store only two bitmaps with approximately 50% of bits set to one.

Suppose the size of RID is eight bytes. Assume there are no bitmap compression. Let n be the number of tuples in the relation, while m is the number of distinct values for the attribute on which the index is being built. The RID list will require 64n bytes, while the bitmaps will require mn bits. Thus if m is less than 64, then bitmap index will require less disk space when compared the traditional Btree. With bitmap encoding this number can be much higher. [3] presents an efficient Encoded Bitmap Indexing for Data Warehouses. But inspite of encodings, bitmaps cannot be used for very high cardinality attribute domain.

2.2 Operations Using Bitmaps

The real strength of bitmaps lies in efficient processing of where clause. The Boolean operations AND, OR, NOT, etc are very fast for bitmaps. Even the COUNT operation can be efficiently performed for the bitmaps[4]. A int array is initialized to contain the number of bits set to one in the entry subscript. Given this, one can loop over the bitmap as an array of integer, and counting the corresponding value in the previous array.

If the where clause has a single term with the equality operator, then the lone bitmap corresponding to this is retrieved. If there are multiple terms, then bitmaps for all of them is got and the resulting bitmap is calculated using the boolean operations among the corresponding bits. [4] provides a modification of bitmaps called Bit-Sliced Indexes for evaluating Range queries over numeric domain.

Joins can be performed efficiently by creating a bitmap index on a table for quantity that involves the join attribute value of the other table.

3 Implementation

For implementing bitmap index we chose Paradise. It already had support for Btree Index, and we extended it to introduce projection index and bitmap index. SHORE, the backend of Paradise, was used for implementing the fact table, storing bitmaps as a tuple, and finally storing key-bitmap pair in a Btree. For the present, we have provided for the bulkloading of the bitmap and projection index. Other operators like selection, join will gradually be implemented.

3.1 Creation of Bitmap Index

We extended the parser so that a user can give a sql command to create either a bitmap index or a projection index. Similarly the sql commands to drop these indexes were provided. The first part of the bulkloading of bitmap index was very similar to that of btree index. Multiple index can built in the same pass. In the following section, building of bitmap index on one attribute is explained. This can be generalized to building multiple bitmap indexes on different attributes.

For building Btree index, the fact table was scanned. The value of the attribute on which the index is being built is saved along with the tuple RID in a temporary file. This file is sorted and the Btree index (storing key-RID pair) is built from it. The temporary file is then deleted. For the bitmap index, during the first scan, the projection index on the attribute is built. Projection Index is nothing but a vertical partitioning of the table. Along with storing the attribute value in a temporary file, the distinct value along with the bitmap number was inserted into a hash table. The bitmap number was used to identify the bitmap corresponding to different attribute value. For example the first attribute value will have a bitmap number of zero. The second distinct attribute value encountered will have the bitmap number of one, and so on. After the first phase, we know the number of tuple and hence the size of bitmaps. In the second phase, the projection index is scanned, and the corresponding bit in the bitmap associated with this particular value for the attribute is set to one. After the scan, we iterate over the hash table, storing the bitmaps into a record, getting the RID for it and then storing this RID with the attribute value as a key in a temporary file. This key-RID pair is sorted on the key and then the Btree index is built as in the previous case. This simple strategy assumes that the bitmaps fit in the memory, which may not be true for large database.

There are two options for the case when the bitmap may not fit in the memory. The first is the horizontal partition of the table and then building bitmaps for each such partition. This divides the relation into small relations and builds bitmap for each one of them. The final bitmap is the concatenation of smaller bitmaps obtained from smaller relations. This can be done in parallel. The other method involves vertical partitioning. This involves multiple scan of the projection index. In this the domain of the attribute is partitioned and complete bitmaps are built for each such partition.

4 Projection Index

As the projection index is built in the intermediate step of building the bitmap index, one can store it in persistence storage for query processing. Attribute value for many tuples can be fit on one page. Query involving only few attribute compared to the total number of attribute can be processed using only the projection index in very few disk IOs. Also if the attributes are of fixed length, one can easily calculate the position of the attribute value of nth record.

5 Conclusion

Eventhough the bitmaps cannot be used for high cardinality attributes, and is difficult to update, they are proving very useful in the read-only environment of data warehousing. We will be implementing other operators using bitmaps, so as to provide a comparative study of traditional btree, bitmaps and projection index.

6 References

1. Breakthrough Performance for Star Schemas. A White Paper by Cheryl Grandy. Dynamic Information Systems Corporation (DISC).

2. The Advancing Art of Indexing. A White Paper by Dynamic Information Systems Corporation (DISC).

3. Wu, Ming-Chuan and Alejandro P. Buchmann. Encoded Bitmap Indexing for Data Warehouses. Technische University Darmstadt, Germany.

4. Patrick O'Neil and Dallan Quass. Improved Query Performance with Variant Indexes. ACM-SIGMOD '97, AZ, USA.

5. Chan, Chee-Yong and Yannis E. Ioannidis. Bitmap Index Design and Evaluation. University of Wisconsin-Madison.

6. Bradley S. Rubin and Jeffrey F. Naughton. Using the SHORE Object-Oriented Database/File System Paradigm for Information Retrieval.

7. Michael J. Carey and et. Shoring Up Persistent Applications. ACM-SIGMOD '9 4, MN, USA.