Subject:
Bitmap indexing and related indexing techniques

Prepared by:
El Ghailani Maher

Supervised by:
Dr. Hachim Haddouti

Spring 2003
1. Introduction  
2. Why Indexing?  
3. Factors that determine the convenient Indexing technique  
4. Criteria to develop a new indexing technique  
5. Bitmap Indexes  
   a. Simple Bitmap index  
   b. Projection Index  
   c. Bit-Sliced Index  
   d. Range-Based Indexes  
   e. Encoded Bitmap Indexes  
6. Advantages and disadvantages of Simple Bitmap Indexes  
7. Comparison of the different Indexes techniques  
8. Conclusion  
9. References
Abstract

The growing interest in Data warehousing for decision-makers is becoming more and more crucial to make faster and efficient decisions. The problem is that most of the queries in a large data warehouse are complex. Therefore, many indexing techniques are created to speed up access to data within the tables and to answer ad hoc queries in read-mostly environments. The traditional B-Tree index technique, used in almost all the relational databases, is still effective for one-dimensional database but it is becoming increasingly ineffective in multidimensional databases because it involves many dimension tables and also attributes and the queries on the data usually involve conditions on small subsets of the attributes. One of the strategies that handle this difficulty is Bitmap indexes. In this paper, I describe the Simple Bitmap Index technique and its related techniques such as Bit-Sliced Index, Range-Based Index and Encoded Bitmap Index and another Indexing technique Projection Index. I also compare those Indexes techniques concerning many parameters. Finally, I draw a conclusion and future enhancement of the Bitmap Indexes.
1. Introduction

Most requests for information from a data warehouse involve dynamic ad hoc queries because users can ask any question at any time for any reason within the base table in a data warehouse. The majority of queries contain a lot of join operations involving a large number of records. Those complex queries could take several hours or days to process because they have to process through a huge amount of data. One of the solutions to speed up query processing is through the use of indexing techniques. Indexes are database objects associated with database tables and created to speed up access to data within the table. They have already existed in the OLTP relational database system but they can not handle large amount of data and complex queries that are common in OLAP systems.

There are lots of indexing techniques that are in use today; however the right choice of a proper index depends on many parameters such as the cardinality data, distribution, and value range. The Bitmap representation is giving an alternate method of the row ids representation. The bitmap is simpler and CPU efficient than row ids when the number of distinct values of the indexed column is low.

First, I will explain the Simple Bitmap Index through an example then I will develop the Bit-Sliced Index Range-Based Indexes, Encoded Bitmap Indexes, and the Projection Index. Second I will show some advantages and disadvantages of Bitmap Indexes by comparing the different Indexes techniques. Finally, I will end up with a conclusion and possible future enhancement.

2. Why Indexing?

Data warehouses are used to store large amounts of data which is often used for Online Analytical Processing (OLAP). However, online decision report needs short response. Therefore, many indexing techniques have been created to reach this goal in read-mostly environments. So the main objective of an indexing technique is to provide the ability to extract data to answer complex and ad hoc queries quickly which is critical for data warehouse applications.

3. Factors that determine the convenient Indexing technique:

Any column that we want to index has its own characteristics that are:

i. **Cardinality data:** The cardinality data of a column is the number of distinct values in the column. There are some indexing techniques that work efficiently only with either low cardinality or high cardinality.
ii. **Distribution**: the distribution of a column is then occurrence frequency of each distinct value of the column. It affects us on choosing the appropriate index type.

iii. **Value Range**: The range of values of an indexed column affects our choice of an index type.

4. **Criteria to develop a new indexing technique**:
   In order to develop a new indexing technique, we have to be concerned with the following characteristics:
   
i. The index should be small and utilize space efficiently
   
   ii. The index should operate with other indexes to fetch the records before accessing raw data.
   
   iii. The index should support ad hoc and complex queries and speed up join operations
   
   iv. The index should be easy to build, implement, and maintain.

5. **Bitmap Indexes**

   Bitmap Indexes were first introduced by O’Neil and implemented in the Model 204 DBMS. This indexing technique is mostly used for typical data warehouse applications, which are mainly characterised by complex query types and read-mostly environments that are more or less static. In data warehouse environments insert, delete or update operations are not very common and, therefore, it is better to build an index which optimises the query performance rather than the dynamic features.

   Bitmap indexes are used by DBMSs to accelerate decision support queries. The main advantage of Bitmap indexes is that complex logical selection operations can be performed very quickly by applying low-cost Boolean operations such as OR, AND, and NOT, thus, reducing search space before going to the primary source data.

   a. **Simple Bitmap Indexes**

   The Simple Bitmap Index consists of a collect of bitmap vectors each of which is created to represent each distinct value of the indexed column. The \( i \)th bit in a bitmap vector, representing value \( x \), is set to 1 if the \( i \)th record in the indexed table contains \( x \). For more detailed definitions:

   - **A Bitmap for a value**: an array of bits where the \( i \)th bit is set to 1 if the \( i \)th record has the value
   - **A Bitmap index**: consists of one bitmap for each value that an attribute can take

   We will illustrate our discussion with a simple example about stock trading. Let us see the following table on stocks traded at different stock exchanges and consider the last column (Exchange).
We notice that stocks are traded at two different stock exchanges at NASDAQ and at NYSE. Indeed, the attribute Exchange has two distinct attribute values. Moreover, we see that our stock example comprises 12 different stocks which are uniquely identified by their record ID given in the first column.

We can represent Stocks and their corresponding trading places by the following simple bitmap:

Example:

NASDAQ: (1 0 0 0 0 1 0 0 0 1 1 1)

NYSE:   (0 1 1 1 1 0 1 1 1 0 0 0)

We need two bitmaps because we have two distinct attribute values for the stock exchange. Each bitmap consists of 12 bit values since our example comprises 12 different stocks.

For instance, the last bit of the bitmap of NYSE is set to 0 because the last stock is not traded at NYSE. However, the second stock is traded at NYSE and, thus, the bit 2 is set to 1. In general, a bit is set to 1 if the stock is traded at the particular stock exchange, and it is set to 0 otherwise. Therefore, we have a straightforward way of describing the stock exchange by means of bitmaps.

A legal question appear now: how do we retrieve data from such a bitmap index? If we make a simple modification of our example and suppose that some stocks are traded at both stock exchanges which could result in the following two bitmaps:

<table>
<thead>
<tr>
<th>Record ID</th>
<th>Ticker Symbol</th>
<th>Trading Volume</th>
<th>Closing Price</th>
<th>Exchange</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AAPL</td>
<td>4675000</td>
<td>36.625</td>
<td>NASDAQ</td>
</tr>
<tr>
<td>2</td>
<td>ABF</td>
<td>642000</td>
<td>24.500</td>
<td>NYSE</td>
</tr>
<tr>
<td>3</td>
<td>AET</td>
<td>368000</td>
<td>72.625</td>
<td>NYSE</td>
</tr>
<tr>
<td>4</td>
<td>CPQ</td>
<td>8968800</td>
<td>51.375</td>
<td>NYSE</td>
</tr>
<tr>
<td>5</td>
<td>DEC</td>
<td>4461100</td>
<td>49.750</td>
<td>NYSE</td>
</tr>
<tr>
<td>6</td>
<td>DELL</td>
<td>2714400</td>
<td>89.750</td>
<td>NASDAQ</td>
</tr>
<tr>
<td>7</td>
<td>HWP</td>
<td>3009300</td>
<td>90.250</td>
<td>NYSE</td>
</tr>
<tr>
<td>8</td>
<td>IBM</td>
<td>7657700</td>
<td>92.500</td>
<td>NYSE</td>
</tr>
<tr>
<td>9</td>
<td>IFMX</td>
<td>3493600</td>
<td>33.000</td>
<td>NYSE</td>
</tr>
<tr>
<td>10</td>
<td>INTC</td>
<td>17694400</td>
<td>65.500</td>
<td>NASDAQ</td>
</tr>
<tr>
<td>11</td>
<td>LGNT</td>
<td>2600</td>
<td>47.250</td>
<td>NASDAQ</td>
</tr>
<tr>
<td>12</td>
<td>MSFT</td>
<td>18238600</td>
<td>91.125</td>
<td>NASDAQ</td>
</tr>
</tbody>
</table>

Figure 1: Stock Trading Example
Example:

NASDAQ: (1 0 1 1 0 1 0 0 0 1 1 1)
NYSE:   (0 1 1 1 1 0 1 1 1 0 1 0)

We notice that the 3rd, 4th and 11th bit in our example are traded at both stock exchange. We simply AND both bitmaps together so that to retrieve this information from our database.

Example:

NASDAQ: (1 0 1 1 0 1 0 0 0 1 1 1)
NYSE:   (0 1 1 1 1 0 1 1 1 0 1 0)

\[ \text{AND} \]

\[ (0 0 \ 1 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0) \]

Given that the 3rd, the 4th and the 11th bits of the resulting bitmap are set to 1, we can know that these stocks are traded at both stock exchanges.

b. Projection Index

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. Projection Index is defined as the materialisation of all values of an attribute in the tuple-ID order. In other words, it is simply a sequence of column values from any table where the ordinal row number of table gives the order of the bitmap index.

Example:

![Figure 2.1: A general Projection Index Example](image-url)
Figure 2.2: An example of the PRODUCT, CUSTOMER and SALE table.

Figure 3 shows the Projection Index on package_type of the PRODUCT table. In general, the queries from a data warehouse retrieve only a few of the table’s columns; therefore, having the Projection Index on these columns reduces extremely the cost of querying because a single I/O operation may bring more values into memory.

c. Bit-Sliced Index

Bit-Sliced Index is a set of bitmap slices which are orthogonal to the data held in a projection index. (i.e. a bitwise vertical partition). Therefore, distinct attribute values of a particular column are represented by means of bitmaps.
Example:

A bit-sliced index based on converting integer values to binary values in order to perform fast logical operations on them since that hardware support directly. We should choose an optimal number of bits per bit-vector in order to represent the whole attribute domain and to occupy minimum space.

d. Range-Based Indexes

The space complexity of the Simple Bitmap index is low for low cardinality attributes but large for high cardinality attributes. Range-Based Index is a simple modification of the bitmap index that handles to some extent this clear weakness. The variation to a Simple Bitmap index is that the bitmap vector is used to represent a range rather than a distinct attribute value as we saw it in our previous example for the attribute Exchange. The most important idea of Range-Based Indexes is to reduce storage overhead by partitioning. That is, attribute values are split into smaller number of ranges and represented by bitmap vectors. Indeed, a bit is set to 1 if a record falls into specified range; otherwise this bit is set to 0.

Example:

We will use the stock trading example. We should first partition the Trading Volume and after that represent it by a Range-Based index. We suppose that a maximum trading volume per day is 20,000,000 shares. Then we divide the attribute Trading Volume into two equal ranges:

\[ [10,000,000, 20,000,000] : \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{bmatrix} \]
\[ [0, 10,000,000) : \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 1 \end{bmatrix} \]

For example, the 10th and 12th stock are traded in a volume greater than 10,000,000 stocks per day since the 10th and the 12th bit of this bitmap vector are set.
The great advantage of a Range-Based index over the Simple Bitmap index is that only a lower number of bitmap vectors need to be stored. Nevertheless, the resulting query process might be longer.

**But how are data retrieved?**

Let see this example:

We suppose that we are interested in all stocks at NYSE that have a trading volume of more than 4 millions shares. Therefore, the two bitmap vectors for the attribute Exchange and the range \([0, 10,000,000]\) are ANDed together:

\[
\begin{array}{c}
[0, 10,000,000]: & (1 1 1 1 1 1 1 1 1 0 1 0) \\
NYSE: & (0 1 1 1 1 0 1 1 1 0 0 0) \\
Candidates & (0 1 1 1 1 0 1 1 1 0 0 0)
\end{array}
\]

There are 7 candidates which are represented by the 1-bit, but we still need to check the value either larger than 4 millions or not.

We notice that Range-Based index needs two search steps instead of only one which is true for Simple Bitmap index. But, one of the great difficulties with this index is to find an optimal partitioning of the range in order to lower the processing time in step 2.

**e. Encoded Bitmap Indexes**

The weaknesses of SBI for high cardinality attributes lead to the suggestion of encoded bitmap indexing which provides the advantage of a drastic reduction in space requirements and also a corresponding performance gain. The main idea of EBI is to encode the attribute domain. We will see the following example:

We assume that we have a fact table SALES with N tuples and a dimension table PRODUCT with 12,000 different products. If we build a simple bitmap index on PRODUCT, It will require 12,000 bitmap vectors of N bits in length. However, if we use encoded bitmap indexing we only need \(\text{ceil}(\log^2 12,000) = 14\) bitmap vectors plus a mapping table which is a very significant reduction of the space complexity.

In this new example we will show how **Huffman encoding** used for reducing the space complexity of bitmap indexes:

We assume that our attribute domain is given by the table \(T\) is \{a,b,c\}. The encoding schema of EBI is stored in a separate table called mapping table and simply encodes the values from a SBI by means of Huffman encoding and therefore reduces the number of bitmap vectors. In particular, we use only \(\text{ceil}(\log^2 3) = 2\) Encoded Bitmap vectors instead of 3 simple bitmap vectors. This means that 2 bits are used to encode the
domain \{a, b, c\}. For example, the attribute value of a is represented by the bit string 100 in the table of the SBI but in the table of EBI the attribute value a is encoded as 00.

![Figure 5: Huffman encoded bitmap index](image)

### 6. Advantages and disadvantages of Simple Bitmap Indexes

**a. Advantages:**

One of the main advantages of bitmap indexes is that logical operations are very well supported by hardware and, thus, the operations are executed quite fast. In addition, both the cost for constructing bitmap indexes and the processing costs are very low.

**b. Disadvantages:**

For high cardinality attributes the space complexity becomes so large that this technique might not be very space efficient.

### 7. Comparison of the different Indexes techniques

<table>
<thead>
<tr>
<th>Indexing Techniques</th>
<th>Characteristics</th>
<th>Advantages</th>
<th>Disadvantages</th>
<th>Implementing Commercial Systems</th>
</tr>
</thead>
</table>
| **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) |
Simple 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  
• 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 |

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 1: Comparison the Indexing Techniques.

8. Conclusion

To sum up, We can say that there is no basic index that is best suited for all applications. Each application has its own specificities. The Bitmap indexing works well in low cardinality but not for high cardinalities. Compressed Bitmap is a promising technique to overcome this problem. Nowadays, there are several fast algorithms for evaluating Boolean operators on compressed bitmaps. Another issue is that we need also some other efficient encoding techniques to lower the number of logical operations.
9. References:

- [2] Ming-Chuan Wu, Alejandro P. Buchmann. Encoded Bitmap Indexing for data warehouses. DVS1, Computer Science Department, Technische Universität Darmstadt, Germany.