Calculating data warehouse aggregates using range-encoded bitmap index.

Kashif. Bhutta
University of Windsor

Calculating Data Warehouse Aggregates Using Range-Encoded Bitmap Index

By
Kashif Bhutta

A Thesis
Submitted to the Faculty of Graduate Studies and Research
through the School of Computer Science
in Partial Fulfillment of the Requirements
for the Degree of Master of Science at the
University of Windsor

Windsor, Ontario, Canada
2002

0-612-75778-1

962148

Abstract

A data warehouse is a database consisting of huge amounts of data collected from different source databases of an organization over a long period of time. Warehouse data are used for analytical purposes to make accurate and timely decisions based on previously integrated facts. Data warehouse is accessed using different kinds of analytical queries. One of the most critical issues is that those queries be responded to quickly and accurately. The size and logical schema of data warehouse systems make it difficult to apply existing query optimizing techniques originally developed for traditional database systems. Indexes are data structures, which help to locate the specific records in the database with minimum number of disk accesses.

Bitmap indexing is a promising technique for data warehousing systems, but space for bitmap indexes is a major problem. This thesis proposes the use of range-encoded bitmap index to calculate aggregates. By using space optimal range-encoded bitmap index for range predicates and aggregates, the need of separate indexes for these operations can be eliminated. The range-encoded index is efficiently used for evaluating range predicates. We are proposing algorithm to evaluate aggregates with the same index that gives equal performance, which was previously achieved by storing a separate index for these operations. This will reduce the space requirements and maintenance overheads considerably without losing performance for aggregates. The proposed indexing scheme is easy to maintain and use the population ratio of 1's in a bitmap to decide if the bitmap has to be scanned from the disk.

Keywords: Data warehouse, query response time, optimization techniques, indexing, bitmap indexes, bitmap encoding schemes, bit-sliced index, range and aggregate predicates

Acknowledgements

There are many people I would like to thank for their help in writing this thesis.

I would like to express my deep appreciation to my supervisor, Dr. Christie Ezeife, for her invaluable comments, encouragement, guidance and support during this thesis work. Without her help, this thesis would not have been completed.

I would also thank my external reader Dr. Maria Cioppa. Her comments contributed a lot to the quality of my thesis. I am very thankful to my internal reader Dr. Shengrui Wang for his pieces of advice and comments, which made me look more critically to the work presented in this thesis. Special thanks also go to Dr. Jessica Chen for finding time to chair my thesis committee.

I would give my sincere gratitude to my beloved wife for her endless love and everlasting support. Her support and cooperation made me able to complete this work.

I love to thank my parents who always stand behind me, love me and support me.

Finally I love to thank my friends from our database group who have been very encouraging and supportive during my thesis work.

Table of Contents

List of Figures

1. INTRODUCTION

Database management systems are widely accepted as a standard tool for manipulating large volumes of data. To enable fast access to stored data according to its content, databases use structures known as indexes. The concept of an index is the same as the index of a book in which the page numbers can be found where a specific word has appeared. Database indexes are used to find the location of a specific record on a disk in order to get it in minimum number of disk accesses. For example, for a certain record, if its disk block address is known, it can be brought into memory for processing with only one disk access rather than with many disk accesses, which occur when a more exhaustive search is used. The disk access is the most expensive operation in a query evaluation process. The goal is to minimize the number of disk accesses either by minimizing the number of accesses for reading the data or by evaluating the query by using indexes only, without accessing the database at all. Some indexes eliminate the need for accessing the database altogether and are able to answer the query by themselves. Bit-sliced index and projection index are the examples of those kinds of indexes.

While indexes are optional, as data can always be located via exhaustive search, they are the primary means for reducing the volume of data that must be fetched and processed in response to a query. Because query performance (response time) is a crucial issue in data warehouse systems, indexing techniques have always been an area of intense research and development. Advances in indexing techniques are primarily driven by the need to support different data models for data warehousing systems [BOS97].

1.1 - Data Warehouse

A data warehouse is a platform with integrated data of improved quality to support many Decision Support System (DSS) and Executive Information System (EIS) applications and processes within an enterprise. Data warehousing improves the productivity of corporate decision-makers through consolidation, conversion, transformation, and integration of operational data, and provides a consistent view of an enterprise. A formal definition of the data warehouse is offered by W. H. Inmon: "A data warehouse is a subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management decisions" [BS97, RMF00]. It is an integrated database established by collecting data from a number of source application databases. A data warehouse contains mostly non-changeable historical data produced over a long time horizon (up to ten years). It is organized around major subjects (entities) of an enterprise, and not around functions [Ez01]. Information is available that allows knowledge workers to make informed decisions, regardless of which database the data was originally collected from. Furthermore, data warehousing should work on integrated data without significantly reducing the performance of a company's transaction systems. Many data warehousing queries involve retrieving enormous amounts of data and joining many large tables. It is very hard to perform complex warehouse operations without suffering in the speed. Warehouse data are used for analytical purposes to make accurate and timely decisions based on previously integrated facts. This process is often called On Line Analytical Processing (OLAP). In the simplest terms, a data warehouse is a very large database in which both current data and historical data are stored. The historical data can be viewed from different dimensions for efficient decision-making.

A data warehouse can be viewed as an information system with the following attributes [BS97]:

- It is a database designed for analytical tasks, using data from multiple applications.
- It supports a relatively small number of users with relatively long interactions.
- Its usage is read-intensive.
- Its content is periodically updated (mostly additions).
- It contains current and historical data to provide a historical perspective of information.
- It contains a few large tables.
- Each query frequently results in a large result set and involves frequent full table scan and multi-table joins.

- It is a database that supports On-Line Analytical Processing (OLAP) for decision support system and is mostly non-changeable collection of data.

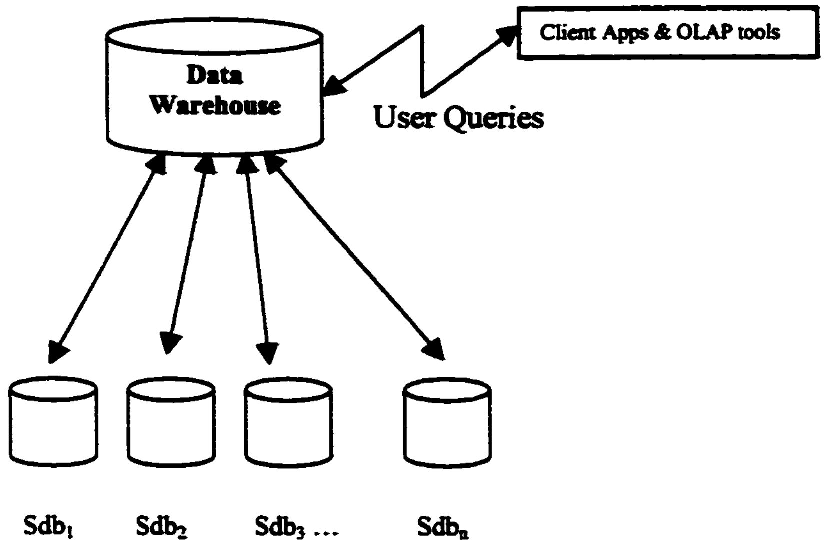


Figure 1.1: An Architecture of a Data Warehouse

Figure 1.1 presents the general idea of a data warehouse, in which the data taken from different data sources is used to establish a data warehouse in a single repository by overcoming software and hardware incompatibilities and after cleaning the data by applying different techniques. The user queries can be posed to that single repository.

As an example, we have two departmental stores belong to the same chain. Each one has its own transactional databases for daily business operations. Furthermore, each of them is using different DBMS (Oracle, FoxPro, Access etc.) and one is using a PC-based system while the other is using Macintosh hardware. For efficient decision making (i.e. inventory on hand), management needs to be able to ask a question such as "*how many BBQ items were sold in each store broken down by the year, by the month and by the day*" or "*how much revenue was generated by ski items in January in each year for the last five years*". To answer these kind of queries, the transactional databases are not enough. The information has to be available from all data sources within a single repository and independent of the software and hardware incompatibilities of the underlying data sources.

In other words, we need a data warehouse. The following is the schema of source database one (Sd1) and two (Sd2) respectively:

Sd1:

Order: *(OrderNo, OrderDate, CustomerId, ProductId, Qty, TotalPrice)*

Customer: *(CustomerNo, Cname, Cadress, Ccity, Cphone)*

Product: *(Pnum, Pname, Pbrand, Psize, Pweight, Packagetype, Unitprice)*

Employee: *(EmpNum, Ename, Eadress, Ephone, Ehiredate)*

Store: *(StoreId, Sadress, NumEmployees, ManagerId)*

Sd2:

Order: *(OrdNum, ODate, CustNum, ProdNum, Qty, TotalPrice)*

Customer: *(CustNo, Cname, Cadress, Ccity, Cphone)*

Product: *(ProdNo, Pname, Pbrand, Psize, Pweight, Packagetype, Unitprice)*

Employee: *(EmpNum, Ename, Eadress, Ephone, Ehiredate)*

Store: *(StoreId, Sadress, NumEmployees, ManagerId)*

Figure 1.2 presents the data warehouse formed by the integration of the above data sources. The data warehouse consists of a central fact table, which is joined by the other tables called dimension tables. Every tuple (fact) in the fact table references a tuple in each of the dimension tables, and may have additional attributes. References from the fact table to the dimension tables are modeled through the usual mechanism of foreign keys. Therefore, each tuple in the fact table is related to one tuple from each of the dimension tables. Vice versa, each tuple from a dimension table may be related to more than one tuple in the fact table. This kind of schema is called star schema. The data from the above two databases are integrated into the star-schema of data warehouse as follows:

Fact Table

Sales: *(OrderNo, ProductID, CustomerID, DateKey, StoreId, Quantity, dollar_amt)*

Dimension Tables

Order: *(OrderNo, OrderDescriptn, OrderDate)*

Customer: *(CustomerNo, Cname, Cgender, Cadress, Cphone)*

Product: *(ProdNo, ProdName, Brand, Size, Weight, Package_type, UnitPrice)*

Date: *(DateKey, Day, Month, Year)*

Store: *(StoreID, StoreAdress, ManagerID, NumEmps)*

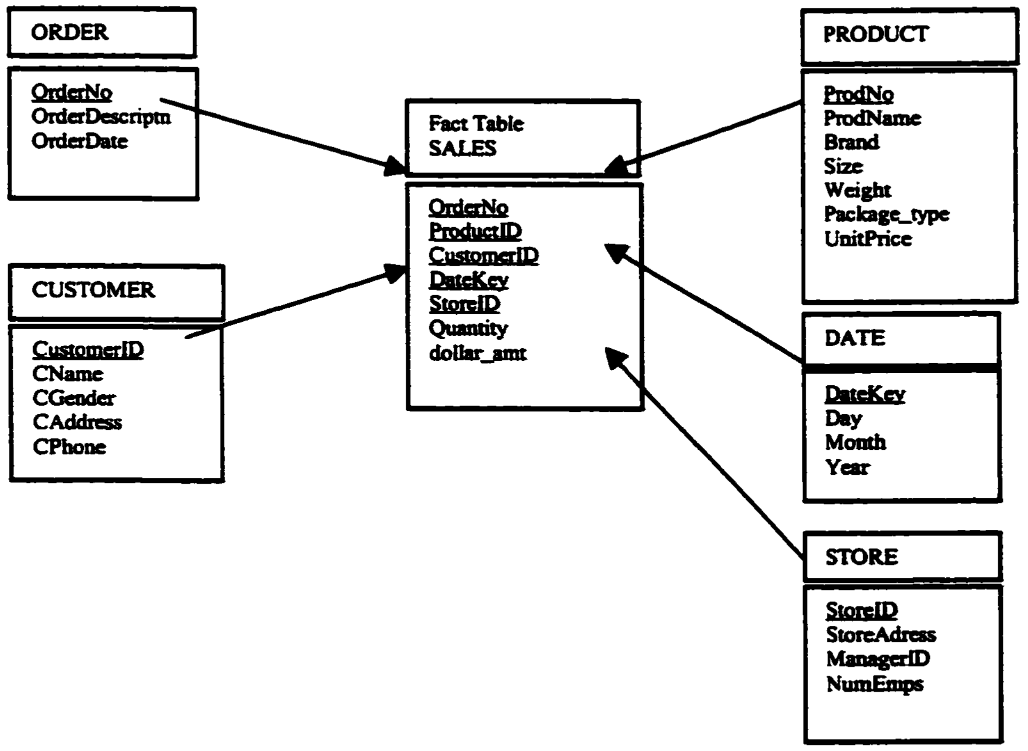


Figure 1.2. An example of star-schema data warehouse with a central fact table (SALES) and several dimension tables

In this example, each tuple of the *SALES* table is a fact of each transaction, while dimensions are attributes about the facts. These facts can be viewed by different dimensions. Examples of dimensions are the product types purchased and the date of purchase. A business question can be asked against this schema much more straightforwardly because we are looking up specific facts (Sale of Items) through a set of dimensions (STORE AREA, PRODUCTS, TIME). It is important to notice that, in the typical star schema, the fact table is much larger than any of its dimension tables. This point becomes an important consideration of the performance issues associated with star schemas.

While the query performance issues of On-Line Transaction Processing (OLTP) systems have been extensively studied and are generally well understood, data warehousing is still evolving as indicated by the growing active research in this area [CD97]. Current database systems, which are optimized mainly for OLTP applications due to their different requirements and workload [Ed95] cannot be used for mostly read only environment of data warehouse technology. There are two techniques used to improve the response time for a warehouse. (1). Pre-calculate the frequent queries and materialize their results as views also called summary tables. A data warehouse is a combination of these summary tables and the base tables. (2). Use indexes to provide fast access to data of tables. The fact that OLAP queries are in most cases ad-hoc queries means that all queries can not be predicted and materialized. It is therefore important to develop sophisticated indexes on base tables to provide adequate performance. The next section presents the existing indexing methods for data warehousing systems.

1.2 - Existing Indexing Methods for Data Warehousing

The existing indexing methods for data warehousing and On Line Analytical Processing (OLAP) systems are classified into four groups. The first class consists of methods that are based on using multidimensional arrays. The second class includes conventional multidimensional indexes originally designed for spatial data. The third class consists of hierarchical methods and finally the methods of the fourth class are based on bitmapped indexes [Sa97]. All four classes are briefly explained.

1.2.1 - Multidimensional Array Based Methods

Logically, the materialized view or OLAP data cube can be viewed as a multidimensional array with the key attributes forming the axis of the array. The ideal indexing scheme for this logical view of the data would be a multidimensional array if the data cube were dense. Any exact or range query on any combination of attributes can easily be answered by algebraically computing the right offsets and fetching the required data. For example, the data in OLAP presented as tuple (shoes, WestTown, 3-July-96, $34.00) will be presented as a cell of an array whose subscripts are [shoes, WestTown, 3-July-96] and the value of that cell is $34.00. The data values are already stored in fixed positions determined by those dimension values, which are made the subscripts of the array and can be determined very quickly.

1.2.2 - Multidimensional Indexes

Another alternative for indexing OLAP data is to apply one of the many existing multi-dimensional indexing methods designed for spatial data. In [Gu84], Gutman presented the idea of multidimensional B-trees and called it R-tree. There are many variations of R-trees such as R+-tree, R*-tree, cube-tree etc. These indexing techniques are mainly used for spatial databases and they are not well explored in the commercial arena of OLAP technology. However, some of these indexing methods offer certain advantages, which can be deployed in indexing OLAP data, with some modifications. One of the key features of several R-tree like indexing schemes is symmetric treatment of all dimensions without incurring the space overhead of the hierarchical indexing methods.

1.2.3 - Hierarchical Indexing Methods

A different approach is used by the hierarchical indexing methods as proposed by Powers and Zanarotti [PZ93] and then by Johnson and Shasha [JS96]. For example, if we have a data warehouse of a chain of stores, in which the information on products is kept, we first build an index tree on the product dimension and store summaries at the product level. Each product value contains a separate index at the store level and stores summaries at the product-store level and so on. Summaries at the store level are kept in a separate index tree on *store*. In general, the number of such index trees can grow exponentially. [JS96] discusses how to cut down the number of trees based on commonly asked queries.

1.2.4 - Bitmapped Indexing Methods

When data is sparse, a good option is not to index the multidimensional data space but to index each of the dimension spaces separately as in bitmapped indexes. This is a popular method used by several vendors, i.e., Sybase IQ [Er95]. Different vendors have different variants of the basic method [OG95]. In the simplest form, a bitmapped index is a B+-tree where instead of storing Row IDs (RIDs) for each key-value at the leaf, a bitmap is stored *(A B+-tree is a balanced tree structure of key values and at the leaf level, it stores the list of RIDs in which this value entry exists)*. Bitmap indexing has been proven to be a promising technique for creating indexes on warehousing base tables. The performance of bitmap indexes is a popular topic of today's research and is a main subject of this thesis.

This thesis focuses on calculating aggregates efficiently with range-encoded bit-sliced indexes. Bit-sliced index is a form of bitmap index. The bitmap index and its variants are explained in next sections followed by the performance issues related to them.

1.3 - Bitmapped Indexes

As explained above, a bitmapped index is a B+-tree where a bitmap stored at leaf-level for each key-value. The database indexes provided today by most database systems use B+-tree structures to retrieve rows of a table with specified values involving one or more indexed columns [Ed95]. The leaf level of the B+-tree index consists of a sequence of entries for index key values. Each key value reflects the value of the indexed column or columns in one or more rows in the table, and each key value entry references the set of rows with that value. A B+-tree is a balanced tree structure of key values and at the leaf level, it stores the list of RIDs in which this value entry exists. For example, if a value 'A' exist in rows 2,3,and 5 of a 10 records table, and 'B' exist in 1,6,and 10, the leaf of 'A' points to a list of 2,3,5, where the leaf of 'B' will point to the list of 1,6,10. In a bitmap index, instead of pointing to the list of RIDs, the leaf level points to a vector of bits. Those bits are set to 1 in which the value exist and the rest of them will be set to zeros. In the above example, the leaf 'A' and the leaf 'B' will point to a bit-vector of 10 bits each and the vector for 'A' will have only bit position 2,3,and 10 set to 1 and for value 'B', the bit positions 1,6, and 10 will be set in its vector.

Traditionally, B+-tree indexes reference each row individually as a RID, a Row ID, specifying the disk position of the row. A sequence of RIDs, known as RID-list, is held in each distinct key value entry in the B+-tree. In indexes with a relatively small number of key values compared to the number of rows, most key values have a large number of associated RIDs and the potential for compression arises by listing a key value once, at the head of what is called RID-list fragment. In a traditional index, each key value is associated with the list of RIDs of tuples having this value for the indexed column. RID lists can be quite long. Moreover, when using multiple indexes for the same table, **intersection, union or complement operations** must be performed on such lists. Therefore, alternative, more efficient implementations of RID lists are important. A bitmap is an alternate method of representing RID-lists in a B+-tree index. Bitmaps are more space-efficient than RID-lists when the number of key values for the index is low. The basic idea is to represent the list of RIDs associated with a key value through a vector of bits. Such vector, usually referred to as a bitmap, has a number of elements equal to the number of tuples of the indexed table. Each tuple in the indexed table is assigned a distinct, unique bit position in the bitmap; such position is called the ordinal number of the tuple in the relation. Different tuples have different bit positions, that is, different ordinal numbers. The i^{th} element of the bitmap associated with the key value is equal to 1 if the tuple, whose ordinal number is *i*, has this value for the indexed column; it is equal to 0 otherwise.

For example, assume a database schema with a table *PRODUCT (pid, brand, size, weight, product_type, price)*, as shown in figure 1.3. A field of the PRODUCT can be used as the indexed column.

Table PRODUCT

product_id	brand	size	weight	package_type	UnitPrice	Ordinal numbers
120	XXX	30	50	A	33.87	001
122	XXX	30	40	B	54.60	002
124	YYY	20	30	A	12.50	003
127	XXX	20	20	A	80.00	004
130	YYY	30	70	C	66.40	005
131	YYY	20	80	C	43.60	006
....						
970	ZZZ	30	80	B	22.30	150

Figure 1.3: A dimension table

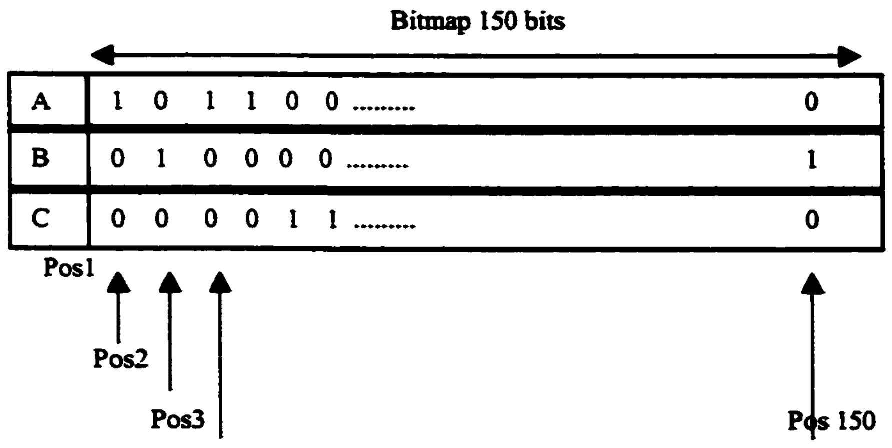


Figure 1.4: An example of bitmap index for package_type

For example, package_type is chosen as the indexed column. The domain of this attribute consists of 'A', 'B' and 'C', means that bitmap index consists of three bitmap vectors, each representing a value of this attribute in its domain. A bit vector is created for 'A'. Another bit vector is created for 'B' and another bit vector is created for 'C' and the three vectors combined make the bitmap index for this attribute. In the *PRODUCT* schema given in figure 1.3, the bitmap index for the *package_type* attribute is a 150 by 3 matrix of bits where the first row represents the index entry (vector) for attribute value 'A' for each of the 150 tuples in the table. Similarly, the second row represents the index entry for attribute value 'B' while the third row represents the index entry for attribute 'C'. The bitmap index for the *package_type* attribute of *PRODUCT* table has been presented in figure 1.4, which consists of three bitmaps of 150 bits each, representing each value of the *package_type* domain.

The first row of the bitmap index is for attribute value 'A' and each tuple value is set to 1 if it has an 'A' value for *package_type* for this tuple, otherwise it is set to zero. For example, tuples 001, 003 and 004 in the database table of figure 1.3 have the value of *package_type* as 'A' and the bitmap entry for 'A' for these tuples are set to 1 while the rest are set to 0.

Similarly, as another example, figure 1.5(a) presents a projection of an attribute *Age* of a relation R, whose values are presented with the duplicates preserved. The values of this column are 0,1,2,...8 and the cardinality of the relation is 12, meaning that there are 12 tuples in this relation.

	$\pi_{Age}(R)$	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	3	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	1	0	0
5	8	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	1	0	0
7	2	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1
9	7	0	1	0	0	0	0	0	0	0
10	5	0	0	0	1	0	0	0	0	0
11	6	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	1	0	0	0	0

(a) (b)

Figure 1.5. Example of a Value-List index (a) Projection of indexed attribute with duplicates preserved (b) Value-List index

The bitmap index of this attribute is a 12 by 9 matrix of bits with one vector of 12 bits each for each of the values in the domain of *Age*. Figure 5(b) presents a bitmap value list index for attribute *Age*. There are nine bitmaps, which is the cardinality of the attribute domain while each bitmap is equal to the cardinality of the relation in length, which is twelve in this case.

The bitmap representation is very efficient when the number of key values in the indexed column is low (as an example, consider a column *sex* of a table *PERSON* having only two values: Female and Male) [OQ97]. In such a case, the number of 0's in each bitmap is not high. In contrast, when the number of values in the indexed column is very high, the number of 1's in each bitmap is quite low, thus resulting in sparsely populated bitmaps. Comparison techniques must then be used. The main advantage of bitmaps is that they result in significant improvement in processing time, because operations such as *intersection*, *union* and *complements* of RID lists can be performed very efficiently by using bit arithmetic. Operations required to compute aggregate functions, typically counting the number of RIDs in a list, are also performed very efficiently on bitmaps. For example, the query *"Find all records of the relation R where the Age = 4"* can be evaluated with the bitmap index of figure 5(b) by scanning the bitmap B^4 only, meaning by scanning 12 bits or 2 bytes from the disk. Another important advantage of bitmap is that they are suitable for parallel implementation [OG95], meaning that the CPU can do the processing on part of index while other tasks can be carried out in parallel to this, which includes the processing of another bitmap from same index. The whole index does not have to be in memory.

Another advantage of bitmap indexes is that complex selection predicates can be computed very quickly, by performing bit-wise *AND*, *OR*, and *NOT* operations on the bitmap indexes. Furthermore, the indexable selection predicates can involve many attributes. As an example, assume that there is a warehouse table *customer* with schema ***CUSTOMER*** **(Name, Lives_in, Works_in, Car, Number_of_children, Has_cable, Has_cellular)**

- *Suppose that we want to select all customers who live in the New York City tri-state area and who drive a car that is frequently purchased in the state in which they live. Then the selection condition is, (Lives_in= "NJ" AND (Car = "Ford Expedition" OR Car = "GMC Suburban")) OR (Lives_in = "NY" AND (Car = "Honda Accord" OR Car = "Ford Taurus")) OR (Lives_in = "CT" AND (Car = "Mercedes 500SL" OR Car = "Cadillac Seville")).*

While conventional indexes in general cannot handle these types of selections easily, bitmap indexes have led to considerable interest in their use in Decision Support Systems (DSS). If bitmaps are available, the above query can be evaluated directly by the intersection and then union of the bitmaps of these attributes. The database will be accessed only to fetch the records after all intersections and unions have been performed and the records in the foundset have been decided. The foundset is a term which is used for a list of record IDs determined to be included in the result set. To understand this example more clearly, consider how the first part of the query *(Lives_in= "NJ" AND (Car = "Ford Expedition" OR Car = "GMC Suburban"))* is evaluated with the B+-tree index on columns *Lives_in* and *Car*. First the query optimizer finds the row IDs which have *Ford Expedition* or *GMC Suburban* in them. Then, it will check if any of those rows have the value "*NJ*" in their *Lives_in* attribute. All the rows, which fulfill this condition, will remain in the result set. On the other hand, if we had bitmap index on both of these attributes, the query was going to be answered in the following steps. First the query optimizer is going to perform OR operation on the bitmaps of *Ford Expedition* and *GMC Suburban* and then just perform an AND operation on the resultant bitmap with the bitmap of *NJ* value of *Lives_in* attribute. This is obvious that the second option is much faster and if the cardinality of indexed attribute is small, it saves a lot of space too.

1.4 - Variations of Bitmap Indexes

The bitmap indexes have different forms and each one is designed for a specific purpose. A brief discussion of each variation is given next to help the reader have a better understanding of bitmap index design.

1.4.1 - Join Index and Domain Index

While traditional table indexes map column values to containing rows in a single table, usually through references to a row identifier, join indexes typically associate column values and rows of two tables. This way, the join index represents the fully pre-computed join. It is a special form of a materialized view. Typical organizations for join indexes include B+-trees or hash indexes organized in any of the following ways: lookup by common join column value listing record identifiers or row Ids (RIDs) in both tables that join with that value; lookup by RID for each row of one table giving a list of RIDs of second table for rows that join with the first row; lookup by (non-join) column value of one table giving a list of RIDs of a second table for rows that join with the rows in the first table having that column value. Variations of these include the use of single column values extended to multiple columns. By using the pre-computed join, it is also possible to access rows of one table through arbitrary column values in a second table using a process that determines rows with given a column value in the second table and then transitively relates those rows through the join index by joining rows of the first table. For example, if we want to maintain a join index of tables *PRODUCT* and *SALE* (figure 1.6), which are having *Product_id* as their joining column, and this is also a primary key of table *PRODUCT*, we will maintain a dynamic list of RIDs of the table *SALE* having the same value in *Product_id* attribute as of *PRODUCT*.

Dimension table
PRODUCT

RID	Product_id	brand	size	weight	package_type
p001	120	XXX	30	50	A
p002	122	XXX	30	40	B
p003	124	YYY	20	30	A
p004	127	XXX	20	20	A
....	130	YYY	30	70	C
	131	YYY	20	80	C
	....				
	970	ZZZ	30	80	B

(a)

Fact table
SALE

Product_id	customer_id	**Ordinal numbers**
120	C25	0001
122	C25	0002
120	C26	0003
120	C28	0004
121	C25	0005
120	C37	.....
131	C40	
120	C70	
122	C25	
....		
130	C40	1800

(b)

Figure 1.6: (a) The dimenssion table PRODUCT (b) The fact table SALE

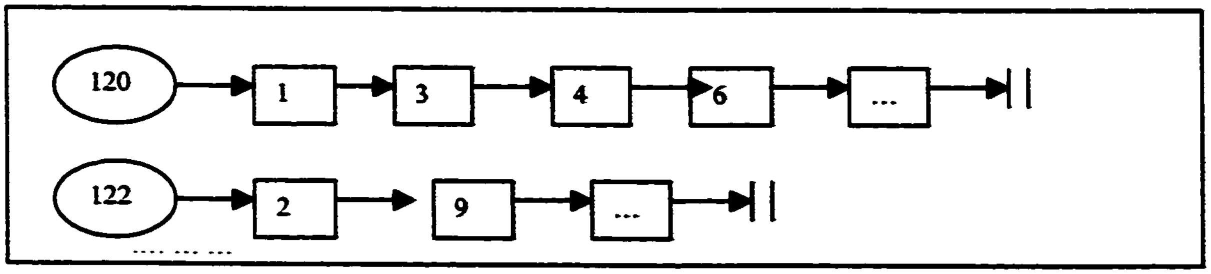


Figure 1.6 (c): An example of Join Index on column Product_id for table SALE and PRODUCT of figure 1.4(a,b)

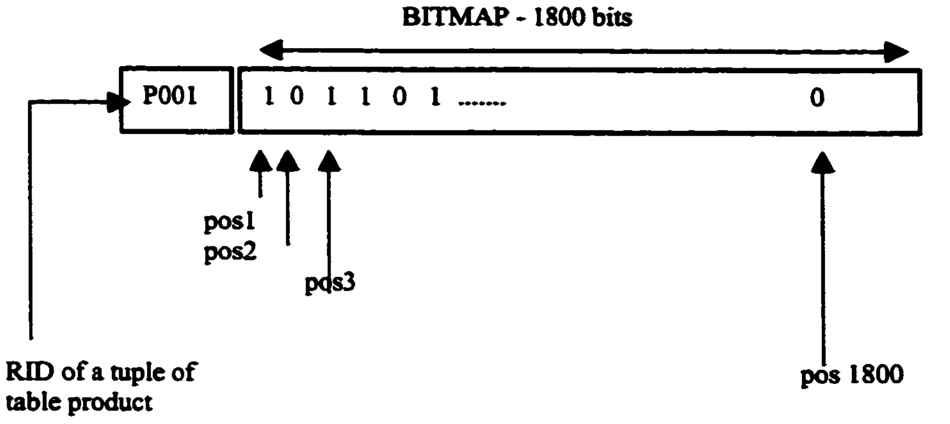


Figure 1.7: An example of a bitmap join index entry of P001 between *PRODUCT* and *SALE* Entry of key value p001 for a bitmap join index allocated on The join between tables PRODUCT and SALES and inverted on RID's of table PRODUCT

As presented in figure 1.6(c), for value 120 of *Product_id* of *PRODUCT*, we have the row numbers 1,3,4,6 ... etc of *SALE* in the dynamic list of joining attribute list, similarly, for value 122 of *PRODUCT*, we have row IDs 2,9 ... etc. of *SALE* having same value in their *Product_id* attribute.

The join index technique aims at optimizing relational joins by pre-calculating them. This technique is optimal when the update frequency is low. In OLAP applications, joins are very frequent and the update frequency is low, so the join index technique can be profitably used here. Join indexes are particularly suited to relate a tuple from a given dimension table to all the tuples in the fact table. Note that the bitmap representation can be combined with the join index technique, thus resulting in a bitmap join index [OG95]. An entry in a bitmap join index, allocated on a fact table and a dimension table, will associate the RID of a tuple *t* from the dimension table with the bitmap of the tuples in the fact table that joins with *t*.

For example, referring to the dimension and fact tables of figure 1.6(a) and 1.6(b) respectively, suppose that a join index is allocated on relations *Sales* and *Customer* for the join predicate *Customer.cusotomer_id* = *Sales.customer_id*. Such join index would list for each tuple of relation *Customer* (that is, for each customer), the RIDs of tuples of *SALES* verifying the join predicates (that is, the sales to the customer). Figure 1.7

presents an example of a bitmap join index on *Product_id* for the first tuple of the dimension table *PRODUCT* to the tuples of fact table *SALE*. The ordinal number of first tuple is p001 and the value of *Product_id* is 120 in that tuple. Figure 7 shows the bit vector for p001 which has 1800 bits in it and exactly those bits are set to 1 where the value 120 appears in 1800 tuples of warehouse fact table. That bit position is 1 in 1800 bits long bit vector and the rest of the bits are 0's. Similarly, according to the same principle, the rest of the ordinal numbers p002 to p030 have their own bit vectors each and they all jointly form bitmap join index for above two tables. Actually, this bitmap index is a matrix of 30 by 1800 bits, since there are 30 products and 1800 tuples in the fact table. There are many compression techniques to reduce the size of these vectors for storage purposes, which is one of the benefits of using bitmaps.

1.4.2 - Projection Index

A projection index is an access structure whose aim is to reduce the cost of projections. The basic idea of this technique is as follows: Consider a column *C* of a table *T*. A projection index on *C* consists of a vector having a number of elements equal to the cardinality of T. The i^{th} element of the vector contains the value of *C* for the i^{th} tuple of *T*. Similar to the bitmap representation, this technique is thus based on assigning the same ordinal numbers to tuples in tables. Determining the value of column *C* for a tuple, given the ordinal number of this tuple, is very efficient. It only requires accessing the i^{th} entry of the vector. When the key values have a fixed length, the secondary storage page containing the relevant vector entry is determined by a simple offset calculation. Such calculation is a function of the number of entries of the vector that can be stored per page and the ordinal number of the tuple can then be calculated. When the key values have varying lengths, alternative approaches are possible. A maximum length can be fixed for the key values. Alternatively, a B+-tree can be used, having as key values the ordinal numbers of tuples and associating with each ordinal number the corresponding value of column C. Figure 1.8 presents an example of a projection index on attribute *Unit_sales*.

Fact table
SALES

Ordinal Number	Product_id	customer_id	Unit_sales
0001	120	C25	50
0002	122	C25	20
0003	120	C26	20
	121	C28	30
	120	C25	70
	130	C37	50
	120	C40	50
....	120	C70	70
	122	C25	20
	....		
1800	130	C40	50

(a)

Unit_sales	Ordinal numbers of index entries
50	0001
20	0002
20	0003
30	
70	
50	
50	
70	
20	....
...	
50	1800

(b)

Figure 1.8. An example of projection index on attribute *Unit_sales*

Figure 1.8(a) is the fact table SALES and figure 1.8(b) presents the projection index on *Unit_sales* which is just a projection of that column with duplicates preserved and saved with the same ordinal numbers as the tuples of the SALES table. Now if a query arrives like *"Find the number of unit sales where Product_id is 120 or 122"*, simply row Ids (ordinal numbers) of those tuples in which the *Product_id* have 120 or 122 can be determined by using any index on *Product_id* say a B+-tree or bitmap join index and then *unit_sales* can be found by accessing the projection index for those ordinal numbers, which are required. The fact table does not need to be accessed. Projection indexes are very useful when very few columns of the fact table must be returned by the query and the tuples of the fact tables are very large or not well clustered. For typical OLAP queries, projection indexes are typically best used in combination with bitmap join indexes. Recall that a typical query restricts the tuples in the fact table through selections on the dimension table. The ordinal numbers of the fact table's tuples satisfying the restrictions on the dimension table are retrieved from the bitmap join indexes. By using these ordinal numbers, projection indexes can then be accessed to perform the actual projection. Note that the actual tuples of the fact table need not to be accessed at all.

For another example, if we have to evaluate a query like *"how many total units are sold where the product_id = 120"*, the ordinal numbers of the fact table can be determined from the bitmap join index of figure 7, where *product_id* is '120' and then projection index of figure 8 can be used to find out the aggregate of total units for that product. Note that the original fact table does not need to be accessed at all, which could have been very time consuming in most of the cases. Similarly, a projection index can be used for the aggregates of total sale amounts *(total_amt)*, total revenue etc.

1.5 - Bit-Sliced indexes

A bit-sliced index (also referred to as binary bit-sliced index) of an attribute is a bit-wise projection of that attribute. The following example explains the bit-sliced index in detail: Consider a table *SALES* which contains rows for all sales that have been made during the past month by individual stores belonging to some large chain. The *SALES* table has a column named *dollar_amt*, which represents for each row the dollar amount received for the sale. Now interpret the *dollar_amt* column as an integer number of pennies represented as a binary number with *N+1* bits. For row with ordinal number n in *SALES* with a non-null value in the *dollar_amt* column, we define a function D(n, i), i = 0,..., N, as follows:

D(n, 0) = 1 if the ones bit for dollar_amt in row number n is on, else D(n, 0) = 0

D(n, 1) = 1 if the ones bit for dollar_amt in row number n is on, else D(n, 1) = 0

...

D(n, i) = 1 if the ones bit for dollar_amt in row number n is on, else D(n, i) = 0

For a row numbered n with a null value in the *dollar_amt* column, we define D(n, i) = 0, for all i. Now for each value i, i = 0 to N, such that D(n, i) > 0 for some row in *SALES*, we define a Bitmap B_i on the *SALES* table so that bit n of Bitmap B_i is set to D(n, i). Note that by requiring that D(n, i) > 0 for some row in SALES, we have guaranteed that we do not have to represent any bitmaps of all zeros. For a real table such as SALES, the appropriate set of bitmaps with non-zero bits can easily be determined at create index time [OQ97]. Thus, in general terms, the Bit-Sliced index on the column *C* of table *T* is the set of all bitmaps B_i as defined in the above example.

The bit-sliced index is further illustrated with the following example.

Example:- Suppose we have an attribute *dollar_amt* of a fact table *SALES* of figure 1.6(b). Figure 1.9(b) presents the projection index on the column. The projection index simply is the projection of the column with the duplicates preserved stored with the same RID numbers.

(Fact Table) Sales

Ordinal No	Product_id	customer_id	dollar_amt
0001	120	C25	970
0002	122	C25	860
	120	C26	950
	121	C28	041
...	120	C25	870
0006	130	C37	859
...	...		
1800	120	C40	272

(a)

Projection Index

dollar_amt
970
860
950
041
870
859
...
272

(b)

Figure 1.9: An Example of projection Index

Figure 10(a) presents the bit-sliced index of the same column. The values in projection index are represented in their binary equivalent and they are stored in slices of bits on disk. Figure 1.10(b) presents those bit-slices separately as bit-vectors. The vectors (b_{10} – b_{15}) are all zeros, so we skip them in the figure. The number of bit-vectors is equal to the memory size of the attribute's data type in bits (16 in this case for a short data type), and the length of each bit vector is equal to the cardinality of the indexed table.

Ordinal No	dollar_amt
0001	0000000001111001010
0002	0000000001101011100
0003	0000000001110110110
0004	0000000000000101001
0005	0000000001101100110
0006	0000000001101011011
...	...
1800	0000000000100010000

(a)

......	b_9	b_8	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	1	1	1	0	0	1	0	1	0
0	1	1	0	1	0	1	1	1	0	0
0	1	1	1	0	1	1	0	1	1	0
0	0	0	0	0	1	0	1	0	0	1
0	1	1	0	1	1	0	0	1	1	0
0	1	1	0	1	0	1	1	0	1	1
..	..	..	..	..	..	..	..	..	..	..
0	0	1	0	0	0	1	0	0	0	0

(b)

Figure 1.10: An Example of Bit-Sliced Index

1.6 - Bitmap Encoding Schemes

The bitmap indexes are stored by different encoding schemes to optimize their performance[CI98]. The most important encoding schemes are the equality-encoding and range-encoding.

Equality Encoding: As has been described earlier, the value-list index is a set of bitmaps, one per attribute value. In other words, if one views this as a two-dimensional bit matrix, the focus is on the columns. If the focus moves on the rows, however, then the index can be seen as the list of attribute values represented in some particular pattern. As there are a large number of possible representations for attribute values (integers in this case), but each attribute value has its unique representation. The following example explains this in detail:

Example of Equality Encoding

Revisiting figure 1.5 for a value list index of an attribute *Age* of relation *R*, it can be observed that the first row in the index has only one bit set to 1 and that is in the bitmap B^3, which corresponds to the value 3, the rest of the bits are set to 0s. So, if there are b_i bits, one for each possible value, the representation of value v_i has all bits set to 0 except for the bit corresponding to v_i, which is set to 1. Similarly, record number 5 has value 8 in its Age attribute, the bitmap B^8 has a 1 in its bit position 5 and rest of the bitmaps B^0 ...B^7 have 0's in that position.

Clearly, an equality-encoded component consists of b_i bitmaps, where b_i is the cardinality of the indexed attribute. In this example, *Age* has cardinality of 9, so there are 9 bitmaps. The Value-List index, which is the simplest, most commonly implemented index, has a single component and is equality-encoded. There are a few problems with saving a bitmap index in an equality-encoded scheme. It does not perform well for the range predicates of the type *Age* <= *4*. To evaluate predicate like this, the bitmaps B0,B1,..B4 have to be scanned and ORed together to get the RIDs in which values are less or equal to four.

	$\pi_{Age}(R)$	B^8	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
1	3	0	0	0	0	0	1	0	0	0
2	2	0	0	0	0	0	0	1	0	0
3	1	0	0	0	0	0	0	0	1	0
4	2	0	0	0	0	0	0	1	0	0
5	8	1	0	0	0	0	0	0	0	0
6	2	0	0	0	0	0	0	1	0	0
7	2	0	0	0	0	0	0	1	0	0
8	0	0	0	0	0	0	0	0	0	1
9	7	0	1	0	0	0	0	0	0	0
10	5	0	0	0	1	0	0	0	0	0
11	6	0	0	1	0	0	0	0	0	0
12	4	0	0	0	0	1	0	0	0	0

(a) (b)

A copy of figure 1.5: Example of a Value-List index (a) Projection of indexed attribute with duplicates preserved (b) Value-List index which is in equality bit encoding scheme

Researchers have proposed another encoding scheme called ***range-encoding*** for bitmaps which can efficiently be used to evaluate range predicates. Range-encoding is explained next.

Range Encoding: There are b_i bits again, one for each possible value. The representation of value v_i has the v_i rightmost bits set to 0 and the remaining bits (starting from the one corresponding to v_i and to the left) set to 1. Intuitively, each bitmap b_{vi} has 1 in all records whose ith value is less than or equal to v_i. Since the bitmap B_{bi-1} has all bits set to 1, it does not need to be stored. So a range encoded component consists of (b_i - 1) bitmaps. The range-encoded index corresponding to the equality-encoded index of figure 1.5 is presented in figure 1.11. There are $b_i = 9$ bitmaps again. For any value vi, (say 3 of the first row), the rightmost bit corresponding to 3 and all bits to the left are set, while the bits to the right of 3 are all 0s in RID 1 of the bitmap index. The value 11111000 standing for a 1 in bits B^3 to B^7 and 0 for bit B^0 to B^2. Since B^8 has all of its bits set, so there is no need to store it.

$\pi_A(R)$	B^7	B^6	B^5	B^4	B^3	B^2	B^1	B^0
3	1	1	1	1	1	0	0	0
2	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	0
2	1	1	1	1	1	1	0	0
8	0	0	0	0	0	0	0	0
2	1	1	1	1	1	1	0	0
2	1	1	1	1	1	1	0	0
0	1	1	1	1	1	1	1	1
7	1	0	0	0	0	0	0	0
5	1	1	1	0	0	0	0	0
6	1	1	0	0	0	0	0	0
4	1	1	1	1	0	0	0	0

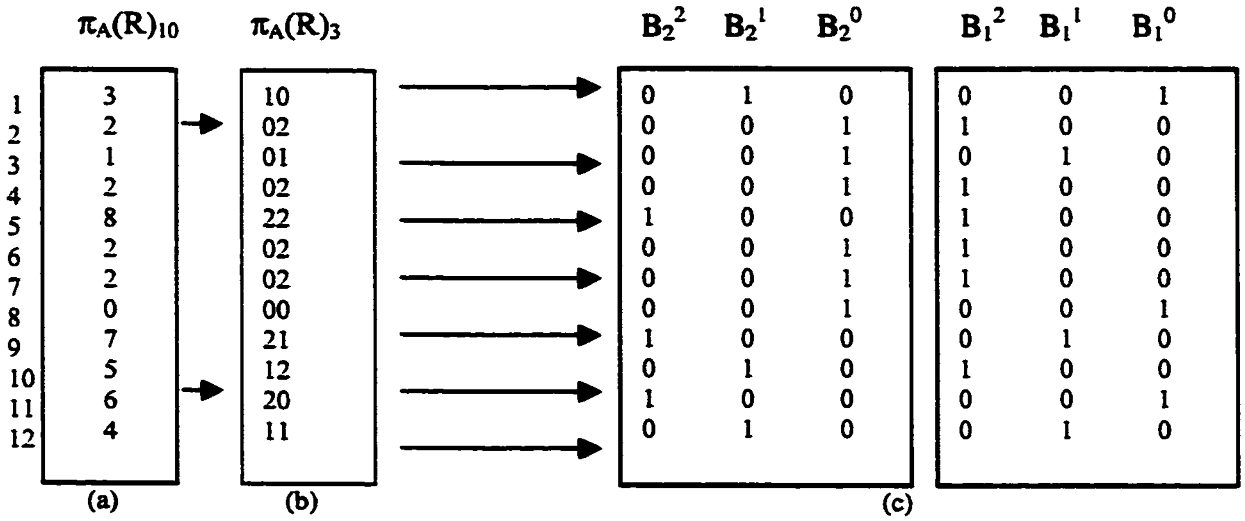
(a) (b)

Figure 1.11: Example of a Value-List index with range-encoding (a) Projection of indexed attribute with duplicates preserved (b) Value-List index with range-encoding

Each encoding scheme has its own advantages. The advantage of range-encoded bitmap over equality-encoded bitmap is that for range queries, fewer bitmaps need to be scanned. For example, to answer a query *"get all tuples with the value of Age less than or equal to 4 (Age <= 4)"*, with the range-encoded bitmap index, only bitmap B^4 is scanned since the answer to the query consists of those tuples where B^4 has a value of 1. On the other hand, answering the same query with the equality encoding-bitmap index requires scanning of $B^0 \ldots B^4$ to obtain the tuples with values 1 in all these bitmaps.

1.7 – Multiple Components Bitmap Index

While the bitmap index is a very attractive alternative, there is a problem if the cardinality of the indexed attribute is high. For example, in the above discussion, if the cardinality of attribute *Age* is 100 (0 – 99) instead of 9, we might end up having 100 bitmaps in an index (one for each value). Similarly if a numeric attribute has its values ranging from 0 to 999, we cannot store 1000 bitmaps for them. Fortunately, the researchers [On87] have provided a very good solution to this problem. For example, if we have values ranging between 000 to 999, for all three-digit numbers, we can have a separate component of bitmap index for each digit. So, instead of having 1000 bitmaps, we will have only 3 components of 10 bitmaps each. In each component, the value can range between 0 and 9; we have to have one bitmap for each expected value. If the values have different base than decimal, each component will have number of bitmaps equal to the base of the digits.



	$\pi_A(R)_{10}$	$\pi_A(R)_3$	B_2^2	B_2^1	B_2^0	B_1^2	B_1^1	B_1^0
1	3	10	0	1	0	0	0	1
2	2	02	0	0	1	1	0	0
3	1	01	0	0	1	0	1	0
4	2	02	0	0	1	1	0	0
5	8	22	1	0	0	1	0	0
6	2	02	0	0	1	1	0	0
7	2	02	0	0	1	1	0	0
8	0	00	0	0	1	0	0	1
9	7	21	1	0	0	0	1	0
10	5	12	0	1	0	1	0	0
11	6	20	1	0	0	0	0	1
12	4	11	0	1	0	0	1	0
	(a)	(b)		(c)				

Figure 1.12. Example of a 2-Component Value-List Index (a) Projection of indexed attribute Values with duplicates preserved. (b)The projection of the same attribute in base-3 numbers (c)Base < 3,3 > Value-List Index.

For example, the value list index of figure 5 can be saved as a two component base-3 index. The decimal 3 is equal to 10 in base-3 numbers and decimal 8 is 22. So if the projection of attribute *Age* is stored as two digits base-3 numbers, the values will be equal to two digit numbers of figure 1.12(b), which is a projection of the same attribute in base-3. A separate component of value-list index for each list of digits is created and presented in figure 1.12(c). Base-3 numbers can have only one out of three digits in them (0,1, and 2). Thus, each component has a maximum of three bitmaps, one for each expected digit. By breaking it from single component of base-10 to two component of base-3 index, we are able to reduce the number of bitmaps from 10 to 6. The index of figure 1.12(c) is referred to as base <3,3> equality-encoded index in the literature, where count of numbers between < and > represent the number of components in the index, and the value of each number represents the base of that component. Figure 1.13 presents the range-encoded base <3,3> index of same attribute whose equality-encoded index has been presented in figure 1.12(c). For range-encoded index, the bitmap for the most significant digit is not needed to be stored since that has all 1's in it, so the index of figure 13 has two range-encoded components of base-3 with two bitmaps each.

B_2^1	B_2^0	B_1^1	B_1^0
1	0	1	1
1	1	0	0
1	1	1	0
1	1	0	0
0	0	0	0
1	1	0	0
1	1	0	0
1	1	1	1
0	0	1	0
1	0	0	0
0	0	1	1
1	0	1	0

Figure 1.13: Base-< 3,3 > Range-Encoded Bitmap Index

By reducing the base and increasing the number of components, we can save some space, but there is a trade-off between space and performance, especially for range queries, so a number of factors should be considered before making a choice.

1.8 – Existing Techniques and Motivation for the Thesis

Various bitmap indexes [OQ97, OG95, On87] have been designed for different query types, including range queries, aggregate queries, and OLAP-style queries. Understanding the space-time tradeoff of the various bitmap indexes is therefore essential for a good physical database design. However, as there is no overall best bitmap index over all kinds of queries, maintaining multiple types of bitmap indexes for an attribute (presently) may be necessary in order to achieve the desired level of performance. Maintaining multiple indexes for an attribute, however, further increases the disk space requirement for data warehouse applications. The bit-sliced index proposed by O'Neil and Quass in [OQ97] gives good performance for aggregate queries, but is not suitable for use in answering range queries. Another disadvantage of this algorithm is that it scans all stored bitmaps of the index whether they make any difference to the solution or not. If a bitmap has all zeros in it and is not stored, the index will have to be recreated each time an update is made to the fact table. For range queries, [OQ97] proposed another type of index called Range-Encoded index. There is still a need for an index which can perform reasonably for both range and aggregate kinds of queries, an index which meets the space constraints and can be used to answer most type of queries without sacrificing performance.

1.9 – Contribution of the Thesis

This thesis presents an algorithm that uses a range-encoded scheme for bitmaps to calculate aggregates. It reduces the disk space overhead by eliminating the need of keeping multiple indexes. On average this algorithm performs better than the algorithm proposed by [OQ97] since it scans a bitmap only if it is needed and the index does not have to be re-built each time a fact table is updated. The algorithm uses the population ratio of 1's in a bitmap, which has already been done in the previous research for range queries, and scans a bitmap only if it affects the solution. In the worst case, this algorithm takes as much space as a regular bit-sliced index [OQ97], and performs as good as the bit-sliced index while it can be used for range queries by the latest techniques proposed in [CI98] and [Wu99], without sacrificing the performance of aggregate operations.

1.10 – Outline of the Thesis

The rest of the thesis is organized as follows: Chapter 2 reviews existing work related to the query evaluation techniques for aggregate and range predicates using Bit-Sliced indexes. Chapter 3 presents a detailed description of the new algorithm for calculating aggregates. Chapter 4 presents the performance analysis and system implementation details and finally chapter 5 gives the conclusion and discusses future work.

2. PREVIOUS/RELATED WORK

In this chapter, the previous research for evaluating predicates with bitmap indexes is reviewed. The algorithms for calculating aggregates and range predicates proposed by O'Neil and Quass [OQ9] are discussed in section 2.1. The algorithm proposed by Chan and Ioannidis [CI98] is discussed in section 2.2 and the algorithm by Wu [Wu99] for evaluating range predicates with tree reduction technique is discussed in section 2.3. Section 2.4 discusses the limitations of these algorithms for evaluating all kinds of queries.

2.1 – Calculating Aggregates and Range with Bit-Sliced Index [OQ97]

O'Neil and Quass [OQ97] have proposed an algorithm for efficiently evaluating sum, count, and average of an attribute using bit-sliced indexes. In most of the cases, bit-sliced indexes are more space efficient and outperform the traditional value-list and projection indexes. When it comes to evaluating a sum, average, or count aggregates like *SELECT SUM(dollar_sales) FROM SALES WHERE <condition>*, bit-sliced indexes give the best

(Fact Table) SALE

Ordinal No	Product_id	customer_id	dollar_amt
001	120	C25	970
002	122	C25	860
	120	C26	950
	121	C28	041
...	120	C25	870
006	130	C37	859
...	123	C22	847
008	120	C40	272
009	125	C32	182
010	130	C10	945
011	123	C28	864
012	120	C40	NULL
013	120	C20	950
014	121	C28	027
015	125	C33	426
016	130	C18	994
017	130	C32	559
018	123	C02	NULL
019	120	C44	283
020	125	C30	782

Figure 2.1 A Fact table with 20 tuples

performance. For the rest of our discussion in this chapter, we are using the fact table *SALE* presented in figure 2.1 with 20 tuples. The attribute *dollar_amt* represents the amount of sale for each transaction in that tuple. We illustrate the algorithm presented by [OQ97] with the following example: suppose we are evaluating a query like *"find the total amount of sale for the products where product_id is 120 or product_id is 122"*.

If the bitmap join index on *product_id* is available, the foundset is simply the union of bitmaps for value 120 and 122. It is assumed that the foundset has already been determined and represented by a bitmap B_f along with the bitmap B_{nn} for all the non-null values in the column. Figure 2.2 below presents the bit-sliced index of the attribute *dollar_amt*.

B_{15}	B_{14}	B_{13}	B_{12}	B_{11}	B_{10}	B_9	B_8	B_7	B_6	B_5	B_4	B_3	B_2	B_1	B_0
0	0	0	0	0	0	1	1	1	1	0	0	1	0	1	0
0	0	0	0	0	0	1	1	0	1	0	1	1	1	0	0
0	0	0	0	0	0	1	1	1	0	1	1	0	1	1	0
0	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1
0	0	0	0	0	0	1	1	0	1	1	0	0	1	1	0
0	0	0	0	0	0	1	1	0	1	0	1	1	0	1	1
0	0	0	0	0	0	1	1	0	1	0	0	1	1	1	1
0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	1	1	0	1	1	0
0	0	0	0	0	0	1	1	1	0	1	1	0	0	0	1
0	0	0	0	0	0	1	1	0	1	1	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	1	1	1	0	1	1	0	1	1	0
0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	1
0	0	0	0	0	0	0	1	1	0	1	0	1	0	1	0
0	0	0	0	0	0	1	1	1	1	1	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	1	0	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0
0	0	0	0	0	0	1	1	0	0	0	0	1	1	1	0

Figure 2.2: Bit-Sliced index on dollar_amt for fact table SALE

The bit-sliced index of figure 2.2 on *dollar_amt* is created by taking each value from the projection of *dollar_amt* column and then converting the projected values to binary numbers. Next, get bit slices from the binary values B_0, $B_1 \ldots B_{15}$. For example from fact table of figure 2.1, the *dollar_amt* value 970 is equal to 0000001111001010 in binary and value 860 is 0000001101011100 in binary. All *dollar_amt* values are stored, and columns representing each bit position (e.g., bit B15,...B0) are the bit-slices. In this example, each bit slice has 20 bits. These values are stored in vertical form of bitmaps which are also called bit slices. The advantage of this method is that either the bitmap of all zero's is not needed to be stored or can be compressed to save a lot of space. The algorithm proposed by O'Neil and Quass [OQ97] to calculate aggregates, and is presented in figure 2.3 and an overview of their algorithm is given next followed by the analysis.

<u>**Overview:**</u> First of all two bitmaps namely B_f and B_{nn} are scanned from the disk. If the tuples of the foundset have null in their *dollar_amt* attribute, there is no need to proceed. So, to evaluate this, an *AND* operation is performed on them and then the result is counted by performing a COUNT operation on the result. If there is no tuple in the foundset which has a non-null value in its *dollar_amt* attribute, the algorithm will return null and finish. If there are some tuples in the foundset which have some values in this attribute and need to be summed up, the algorithm proceed to the next step. The next step is to initialize the *SUM* and scan all bit-slices one-by-one in the *FOR* loop.

```
/* We are given a Bit-Sliced index of dollar_amt, containing bitmaps Bi, i = 0 to N (N = 16),
B_nn for not-null values and B_f for the foundset */

If (COUNT ($B_f$ AND $B_{nn}$) == 0)
        Return null;
SUM = 0.00;
For i = 0 to N
        SUM += $2^i$ * COUNT($B_i$ AND $B_f$);
Return SUM;
```

Figure 2.3: Evaluating SUM with a Bit-Sliced index

It is assumed that B_f stays in memory once read. The *FOR* loop is executed for as many times as there are bit-slices in the index. In each execution of the *FOR* loop, a bitmap B_i is scanned from the disk, it is ANDed with the B_f and then a *COUNT* is performed on the result of that *AND*. That count is multiplied with the appropriate exponent of 2 to get the sum in decimal format. Finally, when all the bitmaps are processed, the calculated *SUM* is returned to the calling module. Next, we will evaluate the performance of this algorithm using the number of bitmap scans and the number of operations on bitmaps when they are brought into memory.

Analysis: The algorithm scans B_f and B_{nn} and then scans all 16 bitmaps of bit-sliced index. Since the assumption is made that B_f remains in memory for the rest of the processing, there is a total of eighteen bitmap scans. There is a total of seventeen *AND* operations performed on those bitmaps and then seventeen *COUNT* operations on the resultant bitmap of that *AND* operation. Since these operations are much less expensive than a bitmap scan, especially, the *COUNT* operation is much less expensive than any other operation, they don't have significant effect on the performance of an indexing scheme. As many compression techniques can be used with bitmaps, especially those bitmaps, which have all zeros or all one's in them, this is an attractive alternative. This algorithm performs best for aggregates (refer to [OQ97] for the comparative analysis of bit-sliced index with other techniques). The main disadvantage of this algorithm is that it scans all the bitmaps whether they are needed or not. For example, it is assumed that if a bitmap has all zero's in it, is not stored. The major disadvantage with this option is that as soon as we insert some values in the table, the bitmaps have to be rebuilt to fulfill the definition and to find bitmap with all zero's in it. Because of this procedure, there is a lot of overhead each time the fact table is updated. To avoid these overheads, all the bitmaps can be stored in most cases. But if a bitmap is stored, the algorithm provides no mechanism for skipping it if that is not needed. The bitmap with all zeros in it results in adding 0 to SUM, so scanning and processing of that bitmap does not affect the solution (SUM) and can be skipped.

O'Neil and Quass [OQ97] also proposed an evaluation algorithm for range predicates of the type (i.e., A op v | op $\in \{<,>,<=,>=\}$). Their algorithm scans all bitmaps of the bit-sliced index and performs 4 operations on each of them, and proved not to be suitable for range queries. For the fact table of figure 14, and bit-sliced index shown in figure 15, it comes to 16 bitmap scans and at least 64 bitmap operations. Since the binary base bit-sliced index does not give a good performance, we skip its details and focus on another algorithm proposed by the same authors to evaluate range in the same publication. They proposed [OQ97] an algorithm to evaluate predicates for range queries with non-binary base bit-sliced indexes. Their algorithm generally works for any non-binary based multiple component bit-sliced index.

B_3^8	B_3^7	B_3^6	B_3^5	B_3^4	B_3^3	B_3^2	B_3^1	B_3^0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	1	0
0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	0	0
1	1	0	0	0	0	0	0	0

B_2^8	B_2^7	B_2^6	B_2^5	B_2^4	B_2^3	B_2^2	B_2^1	B_2^0
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0
1	1	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0

B_1^8	B_1^7	B_1^6	B_1^5	B_1^4	B_1^3	B_1^2	B_1^1	B_1^0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0
1	1	0	0	0	0	0	0	0
1	1	1	1	1	1	1	0	0
1	1	1	1	1	1	1	0	0
1	1	1	1	0	0	0	0	0
1	1	1	1	1	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	1	1
1	1	0	0	0	0	0	0	0
1	1	1	0	0	0	0	0	0
1	1	1	1	1	0	0	0	0
0	0	0	0	0	0	0	0	0
1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	0	0

Figure 2.4: 3- components uniform base-10 range-encoded bitmap index

```
Input:   n is the number of components in the range-encoded index
         < bn, bn-1, ..., b1> is the base of the index. <10,10,10> in our case
         op is the predicate operator, op ∈ {<, >, <=, >=, =, ≠ }
         v is the predicate value
         $B_{nn}$ is a bitmap representing the set of records with non-null values for indexed attribute
         $B_0$ and $B_1$ are the bitmaps of all zeros and all one's respectively

Output:  A bitmap representation of the set of records that satisfies the predicate "A op v"

$B_{GT} = B_{LT} = B_0$;
$B_{EQ} = B_{nn}$;

Let $v = v_n v_{n-1} .. v_1$;
For i = n downto 1 do
    If( $v_i > 0$) then
        $B_{LT} = B_{LT} \vee (B_{EQ}$ OR $B_i^{vi-1})$;
        If($v_i < b_i-1$) then
            $B_{GT} = B_{GT}$ OR ($B_{EQ}$ AND $(B_i^{vi})'$);
            $B_{EQ} = B_{EQ}$ AND ($B_i^{vi}$ XOR $B_i^{vi-1}$);
        Else
            $B_{EQ} = B_{EQ}$ AND $(B_i^{bi-2})'$;
    Else
        $B_{GT} = B_{GT}$ OR ($B_{EQ}$ AND $(B_i^0)'$);
        $B_{EQ} = B_{EQ}$ AND Bio;

$B_{NE} = (B_{EQ})'$ AND $B_{nn}$;
$B_{LE} = B_{LT}$ OR $B_{EQ}$; $B_{GE} = B_{GT}$ OR $B_{EQ}$;
Return $B_{op}$;
```

Figure 2.5: <u>RangeEval Algorithm</u>

Note that the algorithm is developed for a range-encoding scheme. We will evaluate its performance for a 3-components base-10 bit-sliced index on *dollar_amt* of fact table *SALE*, which is presented in figure 2.4. The overview of the algorithm is given next followed by the analysis. The algorithm itself is presented in figure 2.5.

<u>Overview:</u> The algorithm scans B_{nn} and assigns it to B_{EQ}. Next, it breaks the predicate constant v into components of the digits according to same base as its components. For our example, if we have the predicate like *"dollar_amt <= 759"*, the predicate constant is 759 and it will be broken into digits of 7,5 and 9 with a uniform base of 10. Then for each component, the required bitmap is scanned in memory. If a component has more than two bitmaps (it depends on the base of the component, base-10 component has nine bitmaps in it and so on) at-least two bitmaps will be scanned from each component. Depending on the value of *op*, the algorithm performs at least four bitmap operations in each pass of the *for* loop. The *for* loop is executed equal to the number of components.

The algorithm evaluates each range predicate operator by computing two bitmaps: the B_{EQ} bitmap and either the B_{LT} or B_{GT} bitmap depending on the predicate operator. For example if the predicate is ">=", then the result bitmap B_{GE} is obtained by computing the bitmaps B_{EQ} and B_{GT}, steps that involved B_{LT}, B_{LE} or B_{NE} are not required. The final result bitmap that is returned is either B_{LT}, B_{LE}, B_{GT}, B_{GE}, B_{EQ}, or B_{NE} corresponding to the predicate operator "<", "<=", ">", ">=", "=", "≠", respectively.

Analysis: The performance of the algorithm varies with different factors. It primarily depends on the type of predicate being evaluated, the number of components, and the base of each component. We evaluate the algorithm with a 3-component uniform base-10 range-encoded bit-sliced index. Our predicate is *"dollar_amt <= 864"*. As there are 3 components, n = 3, v = 864 and *op* is <=. For this predicate, the algorithm performs **six** bitmap scans and in each pass of the *for* loop, performs at least **three** bitmap operations, which comes to total of 6 bitmap scans and 10 bitmap operations. See [CI98] for a detailed discussion of this example. Although, this is a significant improvement from evaluating aggregates with equality encoded bit-sliced index and the projection index, the algorithm calculates the "≤" and " ≥" type predicates by first calculating "<" and ">" predicates respectively and then *ANDing* this with the results of "=" predicate. This two step process is eliminated by Chan and Ionnadis [CI98], and an optimized range evaluation algorithm is proposed by them. The next section discusses the algorithms proposed by Chan and Ioannidis [CI98].

2.2 – Optimization in Range Evaluation with Bit-Sliced Index [CI98]

Chan and Ioannidis present an improved algorithm to perform the range evaluations with range-encoded bit-sliced indexes. The algorithm to which they called *RangeEval_Opt* reduces the bitmap operations by about 50% and requires one less bitmap scan for a range predicate evaluation. Algorithm *RangeEval_Opt* avoids the intermediate equality predicate evaluation by evaluating each range query in terms of only the "<=" based on the following three identities:

(1). $A < v \equiv A <= v-1$ (2). $A > v \equiv (A<=v)$ (3). $A >= v \equiv (A <= v-1)$

The algorithm has been evaluated next for the same predicate *"dollar_amt <= 864"*. The overview of the algorithm is presented next followed by the analysis. The algorithm is itself presented in figure 2.6.

Overview: The algorithm initializes the resultant bitmap B equal to B_1, which is the bitmap containing all 1's in it. Then, the value of the predicate constant is adjusted depending on if *op* is one of the two < or >=. Then, it executes only the part of the code that is needed. If we are evaluating any of the equality predicates = or ≠, it will execute the second part of the algorithm. If we are evaluating any of the range predicates, the first part of the algorithm will be executed. In either case, it scans only one bitmap from the first component and two bitmaps from subsequent components. In every execution of the *for* loop, it performs two bitmap operations. At the end, it returns the updated bitmap B as resultant, according to the value of *op*.

```
Input:  n is the number of components in the range-encoded index
        < bn, bn-1, ..., b1> is the base of the index. <10,10,10> in our case
        op is the predicate operator, $op \in \{<, >, <=, >=, =, \neq\}$
        v is the predicate value
        $B_{nn}$ is a bitmap representing the set of records with non-null values for indexed attribute
        $B_0$ and $B_1$ are the bitmaps of all zeros and all one's respectively

Output: A bitmap representation of the set of records that satisfies the predicate "A op v"

B = $B_1$;
If (op $\in$ {<, >=} then v = v - 1;

Let v = $v_n v_{n-1} .. v_1$;
If (op $\in$ {<, >, <=, >=}) then
        If($v_1 < b_1 - 1$) then B = $B_1^{v1}$:
        For i = 2 to n do
                If ( $v_i \neq b_i - 1$) then B = B AND $B_i^{vi}$ ;
                If ( $v_i \neq 0$) then B = B OR $B_i^{vi-1}$ ;
Else
        For i = 1 to n do
                If ( $v_i = 0$) then B = B AND $B_i^0$ ;
                Else If ( $v_i = b_i - 1$) then B = B AND $(B_i^{bi-2})'$;
                Else B = B AND ($B_i^{vi}$ XOR $B_i^{vi-1}$);

If (op $\in$ { >, >=, $\neq$ } then
        Return (B' AND $B_{nn}$);
Else
        Return (B AND $B_{nn}$);
```

Figure 2.6: RangeEval_Opt Algorithm

Analysis: The algorithm *RangeEval_Opt* is evaluated for the predicate "A <= 864". Since *op* is <=, the first part of the *if* statement will be executed. The algorithm scans a bitmap from the first component and assigns it to the resultant. Then, the *FOR* loop is executed for the rest of the components and in each execution of the *FOR* loop, the algorithm performs two bitmap scans and two operations. For this example, there are three components, so the *FOR* loop will be executed two times and will perform four operations during the execution of the *FOR* loop. The total bitmap scans come to five and the total operations are five too. This algorithm evaluates the queries with less bitmap scans and far less operations on them than does the algorithm *RangeEval*.

Increasing the number of components of a bitmap means that we have a small base for components and each component consists of a small number of bitmaps. For example, a base-3 component will have only three bitmaps in it since there are only three digits in base-3 numbers. In range-encoding scheme we do not store the bitmap for the highest digit. Thus, a base-3 component will have only two bitmaps for bits b_0 and b_1. Similarly, a base-10 component has 9 bitmaps in it since decimal numbers have 10 digits and we do not store the bitmap for digit 9. Fewer components implies that we have more bitmaps in each component and we will have to scan less bitmaps to evaluate a range predicate, though each component will require more space. The range-encoded base<10,10,10> index of figure 17 can be stored as a range-encoded base<2,2,2...2> 16 component bitmap index. Each component will have only one bitmap in it since there is no need to store a bitmap for highest digit, which is 1 in this case. By expanding it to a 16 component index of one bitmap each, we can reduce the number of bitmaps from 30 to 16. But there is a price for this space saving, as there will be more bitmap scans for evaluating a range predicate. Chan and Ioannidis [CI98] provided a framework for getting an optimized solution for tradeoff between time and space. Evaluating the predicate "*dollar_amt* < 864" with a 16 component bitmap index of uniform base two will require one bitmap scan from each component and one operation on them which makes it a total of 16 bitmap scans and 16 bitmap operations and might not be considered an acceptable performance in most of the cases. This number can be reduced by applying the tree reduction technique on this index. The tree reduction technique was proposed by

Wu [Wu99] to reduce the bitmap scans, extending opportunity for space optimized range-encoded indexes for range queries. The tree-reduction technique is discussed next.

2.3 – Using Tree Reduction for Query Evaluation with Bit-Sliced Index [Wu99]

Further improving upon *RangeEval-Opt*, Wu [Wu99] proposed an execution tree reduction technique. Using the same example, we are given a 3-component range bit-encoding with decimal base on attribute "*dollar_amt*" and the predicate "*dollar_amt* <= *864*". Using algorithm *RangeEval-Opt* to evaluate the predicate results in the execution tree as shown in figure 2.7(a). Suppose that for some certain running state of a database,

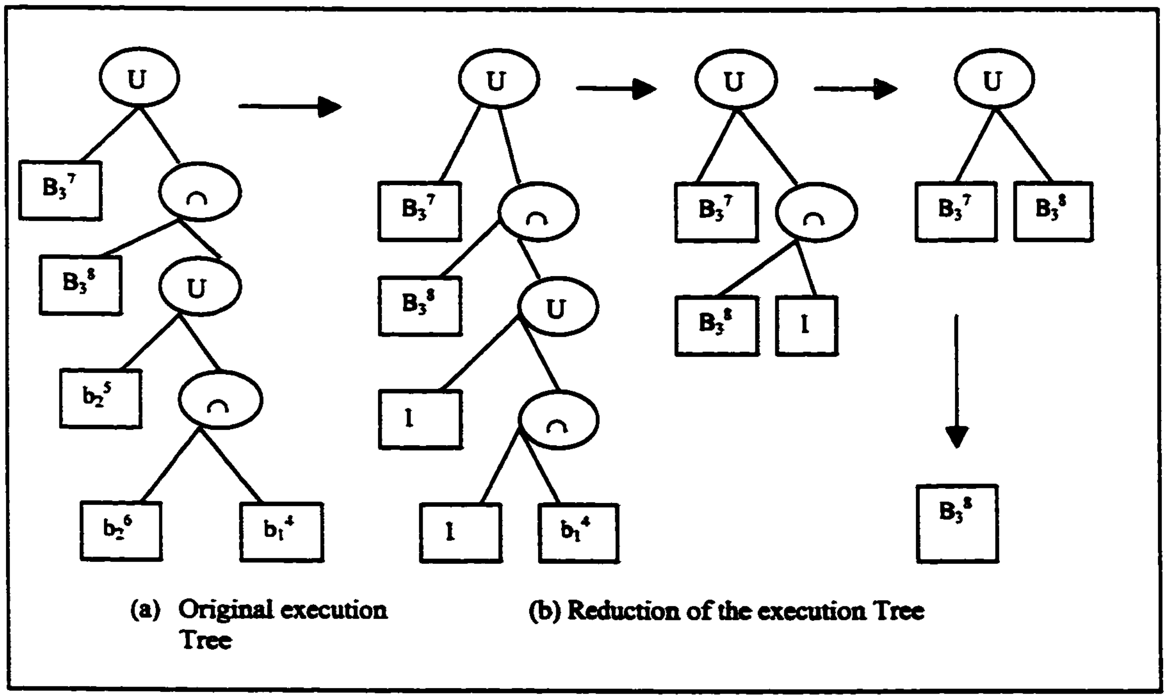

U
∩
B_3^7
B_3^8
b_2^5
b_2^6
b_1^4
1
(a) Original execution Tree
(b) Reduction of the execution Tree

Figure 2.7: Transformation of execution tree for dollar_amt <= 864

the second digit of all the values of *dollar_amt* is no larger than 5, i.e., in the component-2 of the index, the bit vectors b_2^5, b_2^6, b_2^7 and b_2^8 are all set to "1". By replacing the corresponding bit vectors with '1'-vectors, we have the first tree in figure 2.7(b). By applying $x \,.\, 1 = x$ (identity law) and $x + 1 = 1$ (domain law) of Boolean algebra, the execution tree is reduced down to one node, i.e., instead of 5 bitmap scans plus 4 logical operations, only 1 bitmap (the bit vector b_3^8 of component-3) is read.

In order to be able to apply such a reduction, the information about the percentage of population of each bit vector is needed. Without much extra cost, this information can be computed at the time of index creation and can be synchronized every time the index is changed [Wu99]. The revised version of *RangeEval-Opt*, which suppresses some unnecessary bitmap scans as described above, is shown in figure 2.8. An overview of this algorithm is given next followed by an analysis.

```
Input:  n is the number of components in the range-encoded index
        < bn, bn-1, .., b1> is the base of the index. <10,10,10> in our case
        op is the predicate operator, $op \in \{<, >, <=, >=, =, \neq\}$
        v is the predicate constant
        $B_{nn}$ is a bitmap representing the set of records with non-null values for indexed attribute
        $B_i^j$ denotes the j-th bit vector of i-th component and for a bit vector $B_i^j$, $\theta(B_i^j)$ denotes the
        percentage of 1's in $B_i^j$

Output: A bitmap representation of the set of records that satisfies the predicate "A op v"

B = 1;
If (op $\in$ {<, >=} then v = v - 1;

Let $v = v_n v_{n-1} .. v_1$;
If (op $\in$ {<, >, <=, >=}) then
        If (($v_1 < b_1 - 1$) AND ($\theta(B_1^{v1}) \neq 1$)) then $B = B_1^{v1}$;
        For i = 2 to n do
        If (( $v_i \neq b_i - 1$) AND ($\theta(B_i^{vi}) \neq 1$)) then B = B AND $B_i^{vi}$ ;
        If (( $v_i \neq 0$) AND ($\theta(B_i^{vi-1}) \neq 1$)) then B = B OR $B_i^{vi-1}$ ;

Else
        For i = 1 to n do
          Switch (vi)
                Case $v_i = 0$: if ($\theta(B_i^0) \neq 1$) then B = B AND $B_i^0$ ;
                Case $v_i = b_i - 1$: if ($\theta(B_i^{bi-2}) \neq 1$) then B = B AND $(B_i^{bi-2})'$
                                Else return (B = 0);
                Case 0 < vi < bi - 1:
                        If (if ($\theta(B_i^{vi}) \neq 1$) AND if ($\theta(B_i^{vi-1}) \neq 1$) then B = B AND ($B_i^{vi}$ XOR $B_i^{vi-1}$);
                        Else if ($\theta(B_i^{vi-1}) = 1$) then return (B = 0)
                        Else B = B AND $(B_i^{vi-1})'$;
          End Switch

If (op $\in$ { >, >=, $\neq$ } then Return B' AND $B_{nn}$;
Else Return B AND $B_{nn}$;
```

Figure 2.8: Algorithm RangeEval_Opt With Tree Reduction Technique

Overview: The improved algorithm presented by Wu [Wu99] performs as good as the original algorithm in the worst case. But it considers the situation in which we end-up scanning and performing on an unnecessary bitmap, which has all 1's in it. The population ration of 1's can be used to decide if a bitmap should be scanned from the disk at all. The algorithm proposes a very small change to *RangeEval_Opt*, it processes a bitmap only if it does not have all 1's in it and passes the condition of *if* statement (`if` ($\theta(B_i^{vi}) \neq 1$) before scanning it, where $\theta(B_i^{vi})$ is the ratio of 1's in the bitmap. There is no need for scanning and performing an operation with a bitmap of all 1's, since the same results can be obtained by applying the identity law and domain law as described above.

Analysis: If the range predicate "*dollar_amt <= 864*" is evaluated by the tree-reduction using the 3-component base-10 index of figure 15. We make the assumption that all the digits in component 3 are less or equal to seven for some state of the data warehouse fact table. So this query is evaluated when B_3^8 and B_3^7 have all 1's in them. The algorithm scans the B_1^4 and assigns it to B (the resultant bitmap). Now in the execution of *FOR* loop, for component 2, bitmaps B_2^6 and B_2^5 are scanned and one *AND* operation and one *OR* operation is performed on them respectively. Next, for component 3, the ratio of 1's in bit vectors B_3^8 and B_3^7 is checked and it is found that both of them have all 1's in them, so there is no need to scan them. The result has already been evaluated. Scanning these two bitmaps has no effect on the resultant bitmap B. By using this technique, we are able to answer the query by scanning 3 bitmaps and performing only two operations on them. However, in the worst case, when B_3^8 and B_3^7 are also scanned, the algorithm gives the same performance as *RangeEval_Opt*.

2.4 – The Limitations of the Existing Techniques

The bit-sliced index proposed by [OQ97] for aggregate calculation is known for the best performance when calculating sum, average, and count aggregates although it scans all the bitmaps of a bit-sliced index if stored. For evaluating the range predicates, they have proposed range-encoded bit-sliced index with multiple components. This scheme is improved significantly by Chan and Ioannidis [CI98] with a reduction in number of bitmap scans and operations on them and is further improved by Wu [Wu99] with the use of tree reduction technique. The two later techniques address the problem of evaluating the range predicates and require that the bit-sliced index is to be available in range bit-encoding. It is assumed that the regular equality-encoded bit-sliced index is also stored and can be used for aggregates using the algorithm proposed by [OQ97]. There is an advantage of having specialized bitmaps for certain operations, e.g., having bit-sliced bitmap index for aggregate queries and range-encoded bitmap index for range queries. The space and maintenance overheads remain a problem for these techniques and having multiple indexes worsens this problem. The other disadvantage of calculating aggregates with bit-sliced index [OQ97] is that either the bitmap is not stored at all if it has all zero's in it or it will be scanned if it is there. In the case where the bitmap is not stored, the index has to be rebuilt each time the fact table is updated. We are proposing an algorithm to calculate aggregates with the range-encoded bit-sliced index that results in an improvement in performance without increase in space. Since, the bit-sliced index is in range-encoded form, it can efficiently be used for the latest algorithms to answer range queries. The index does not have to be rebuilt each time the fact table is updated. The proposed method will also eliminate the need for maintaining separate indexes for range and aggregate predicates, which will reduce the space requirements and maintenance overheads considerably.

3. CALCULATING AGGREGATES USING RANGE-ENCODED SCHEME

There are two challenges faced by data warehouse designers. First, the space for storing the bitmap index should be reduced so that maintenance cost should be as low as possible. Secondly, low time complexity is desirable, so that the query should be answered as quickly as possible. In this thesis, an algorithm is proposed to perform aggregate operations with the use of a range-encoded bitmap index. The disk read/write is the most expensive operation in query evaluation. Our algorithm improves the response time by reducing the number of bitmap scans significantly. We are proposing a space optimized range-encoded bitmap index, which can be used by the latest techniques for evaluating range predicates. We will calculate both aggregates and range predicates in the evaluation of this technique and will compare its performance with the other techniques.

For our discussion in this chapter, we are using the fact table *SALE* first presented in figure 2.1. Since *dollar_amt* is a short data type with memory size of two bytes and 16 bits, that is the reason the bit-sliced index of figure 2.2 has 16 bitmaps in it. If we develop a range-encoded uniform base-2 bit-sliced index on the same attribute, that will have 16 components in it, because any value of short data type can have up to sixteen digits and each digit will be either 0 or 1. So, for a space optimized range-encoded bit-sliced index of binary base, we need to have sixteen components of two bitmaps each. As they are stored in a range-encoded scheme, we do not have to store the bitmap for the most significant digit, which is 1 in this case. Thus, we only need to store the bitmap for digit 0 in each component. The index is presented in figure 3.1. As obvious, this index consumes the same space as the bit-sliced index of figure 2.2. Although it looks like as if all bitmaps are stored, in practice, the bitmaps of all 1's do not need to be stored since they can either easily be generated within memory without incurring significant cost or can be compressed to a mere four bytes with any compression technique. The difference is that this is a range-encoded bitmap index of 16 components. Note that this index is exactly the complement of the bit-sliced index of figure 2.2.

B_{16}^0	B_{15}^0	B_{14}^0	B_{13}^0	B_{12}^0	B_{11}^0	B_{10}^0	B_9^0	B_8^0	B_7^0	B_6^0	B_5^0	B_4^0	B_3^0	B_2^0	B_1^0
1	1	1	1	1	1	0	0	0	0	1	1	0	1	0	1
1	1	1	1	1	1	0	0	1	0	1	0	0	0	1	1
1	1	1	1	1	1	0	0	0	1	0	0	1	0	0	1
1	1	1	1	1	1	1	1	1	1	0	1	0	1	1	0
1	1	1	1	1	1	0	0	1	0	0	1	1	0	0	1
1	1	1	1	1	1	0	0	1	0	1	0	0	1	0	0
1	1	1	1	1	1	0	0	1	0	1	1	0	0	0	0
1	1	1	1	1	1	1	0	1	1	1	0	1	1	1	1
1	1	1	1	1	1	1	1	0	1	0	0	1	0	0	1
1	1	1	1	1	1	0	0	0	1	0	0	1	1	1	0
1	1	1	1	1	1	0	0	1	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	0	0	0	1	0	0	1	0	0	1
1	1	1	1	1	1	1	1	1	1	1	0	0	1	0	0
1	1	1	1	1	1	1	0	0	1	0	1	0	1	0	1
1	1	1	1	1	1	0	0	0	0	0	1	1	1	0	1
1	1	1	1	1	1	0	1	1	1	0	1	0	0	0	0
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	1	1	1	0	1	1	1	0	0	1	0	0
1	1	1	1	1	1	0	0	1	1	1	1	0	0	0	1

Figure 3.1: 16-component Bit-Sliced index on dollar_amt fact

3.1 – Evaluating Aggregates

When it comes to evaluating a sum, average, or count aggregates like *SELECT SUM(dollar_sales) FROM SALES WHERE <condition>*, bit-sliced index gives the best performance of the techniques evaluated [OQ97]. We illustrate our algorithm for the same kind of aggregate so it will be helpful for comparative analysis later. In the rest of the discussion, we also assume that the foundset has already been determined and represented by a bitmap B_f along with the bitmap B_{nn} for all the non-null values in the column.

The overview of the algorithm is presented next followed by a detailed comparative analysis. The algorithm itself is presented in figure 23.

```
Input:  We are given a n-component base-2 Bit-Sliced index for an atribute
        n is the number of components in the range-encoded index ( n= 16 in this case)
        $B_f$ is the bitmap of foundset, $B_{nn}$ represent the bitmap of non-null values
        $B_i^j$ denotes the j-th bit vector of i-th component and for a bit vector $B_i^j$, $\theta(B_i^j)$ denotes the
        percentage of 1's in $B_i^j$

Output: The sum of the attribute values satisfying the found set


If ( COUNT ($B_f$ AND $B_{nn}$ ) == 0 )
        Return null;

CNT = COUNT($B_f$);
SUM = 0.00
For i = 1 to n
        If ($\theta(B_i^0) \neq 1$) then SUM += $2^{(i-1)}$ * ( CNT  - COUNT ($B_i^0$ AND $B_f$));

Return SUM;
```

Figure 3.2: <u>Calculating aggregates with range-encoded bitmap</u>

<u>**Overview:**</u> First of all two bitmaps B_f and B_{nn} are scanned from the disk. It performs an *AND* operation on them. If all the tuples in the foundset have null values in their required column, the algorithm will return null and finish. Next it performs a *COUNT* operation on the foundset and stores the result in a variable named *CNT*. It initializes the *SUM* and goes into the *FOR* loop. The *FOR* loop will be executed for the same number of times as there are components. There are 16 components so the *FOR* loop will be executed 16 times. In each execution of the loop, the bitmap is scanned only if it does not have all 1's in it or its population ratio is not 1. In other words, if it satisfies the condition of the *if* statement, provided in the single statement of the *FOR* loop. If the bitmap has all 1's in it, it is not needed to be scanned and operated upon; since that bitmap has zeros in all of its attribute values for that digit, it does not affect the value of *SUM*. If a bitmap is scanned, it will be *ANDed* with the foundset B_f, which stays in memory for the rest of the processing after it has been scanned from the disk the first time. Then a *COUNT* operation is performed on the result of that *AND* operation. This count is subtracted from the variable *CNT* and the resultant figure is multiplied with appropriate exponent of 2. The *SUM* is then returned to the calling module.

Analysis: The algorithm must scan two bitmaps namely B_f and B_{nn}. They are ANDed and a *COUNT* is performed on the result of the *AND* operation on those two bitmaps. If the result is not zero, the algorithm proceeds. It counts the number of 1's in the foundset by performing a *COUNT* on the bitmap B_f. The result is stored in a variable CNT. The algorithm executes the *FOR* loop for the number of components. In the worst case, it will scan all the bitmaps in each execution of the loop. If all the bitmaps are scanned, there will be 16 bitmap scans in the loop. In each iteration of the *FOR* loop, if a bitmap is scanned that will be *ANDed* with the B_f, then the *COUNT* operation is performed on the result of *AND* operation between B_f and B^0_i. For each iteration of the *FOR* loop, the algorithm scans one bitmap and performs two bitmap operations. For a short data type, the algorithm scans 18 bitmaps and performs 17 *AND* operations on their pairs and it performs 18 *COUNT* operations on those bitmaps (worst case scenario). Any operation on the bitmap is much less expensive than a bitmap scan and a *COUNT* operation is even less expensive than an *AND* operation. So with the total of 18 bitmap scans and 35 bitmap operations, this algorithm gives as good performance as one proposed by [OQ97].

Practically, it is much less likely that, for any data type, all the bitmaps will have to be scanned from the disk. In our example, for a short data type and the bit-sliced index of figure 20, the bitmaps of last 6 components have all 1's in them. So it will perform only 12 bitmap scans and 23 bitmap operations. This is a significant improvement over the one proposed by [OQ97] if all bitmaps are stored. If all bitmaps are not stored, the proposed algorithm gives the same performance as that algorithm. So there is no loss in performance for aggregates. Also, the proposed index is a space optimized range-encoded index and can be used for the latest evaluation techniques for the range queries.

3.2 – Evaluating Range

Next, we will see how a range query like *"dollar_amt <= 864"* can evaluated with the proposed index. For a range query like this, the algorithm of [Wu99] can be used. The bitmaps of last 6 components have all 1's in them. There are only 10 bitmaps which will pass the criteria of the *if* statement, so only 10 bitmaps will be scanned and the algorithm will perform only one operation on each bitmap. So the above query can be evaluated with 10 bitmap scans and 10 bitmap operations which is still within the reasonable bounds as compared to the bitmap index of base <10,10,10> which has 30 bitmaps in it and takes 5 bitmap scans and 5 bitmap operations. Considering that with two indexes, one bit-sliced index with 10 bitmaps and second range-encoded 3-component <10,10,10> with 27 bitmaps in it, the aggregates can be evaluated with 18 bitmap scans and 34 bitmap operations, and the range can be evaluated with 5 bitmap scans and 5 bitmap operations, while with our single index of 10 bitmaps, the aggregates can be evaluated with 12 bitmap scans and 23 bitmap operations on them and the range can be evaluated with 10 bitmap scans and 10 bitmap operations. The choice of another index is still there for optimizing range queries with higher base index.

4. PERFORMANCE ANALYSIS

4.1 – Performance Analysis from Theory

Consider a data warehouse having a fact table SALE, in which data is integrated for all transactions of all stores during the first week of November of last year, and a dimension table PRODUCT having the details of all products being sold across all stores. The fact table and dimension table have the following schema and are represented with some data in them in figure 4.1.

Sales: *(OrderNo, ProductID, StoreId, Quantity, dollar_amt)*

Product: *(ProdNo, ProdName, Brand, Size, Weight, Pkg_type, UnitPrice)*

(a) Dimension table PRODUCT

RID	ProdNo	ProdName	Brand	Size	Weight	Pkg_type	UnitPrice
p001	120	Golf Clubs	30	50	-	A	150
p002	122	Golf Shoes	30	40	-	B	80
p003	124	YYY	20	30	-	A	10
p004	127	XXX	20	20	20	A	20
....	130	ZZZ	30	70	30	C	40
	131	JJJ	20	80	50	C	35
p007	970	SSS	30	80	10	B	200

(b) Fact table SALE

Ordinal numbers	OrderNo	ProductID	StoreID	dollar_amt
001	R001	120	S 25	970
002	R002	122	S25	860
003	R420	120	S26	950
004	R010	120	S28	041
005	R300	121	S25	870
	R130	120	S37	859
	P003	131	S40	847
	P004	120	S70	272
	P200	122	S25	182
...	P210	970	S26	945
	P220	124	S28	864
	P320	122	S70	NULL
	P110	127	S26	950
	R015	131	S40	027
	R400	120	S37	426
	P190	131	S25	994
	P500	970	S40	559
	P600	131	S37	NULL
	P700	127	S70	283
020	P310	130	S40	782

Figure 4.1: (a) A dimension table for product (b) A fact table contains data of transactions for November

Now the administration has to make a decision that if they want to keep on-hand inventory for the golf items for coming winter. Golf is a summer support but a new trend has arisen of playing golf indoor during winter. The management needs to know if they want to keep inventory for the golf during the month of November this year and wants to find out how much revenue was generated by the golf items last year. So the expected question is *"how much total revenue was generated by all stores where the item name is Golf Club or Golf Shoes"*. As it is obvious that this kind of arbitrary queries are not predictable and the results can not be stored in summary tables for them, but with the use of sophisticated indexes, such kind of queries can be evaluated very easily.

We assume that a bitmap join index exist between the two tables through the joining attribute of *ProductID*. If the above query is translated into an SQL query, it looks like the following

"SELECT sum(S.dollar_amt) "Golf Revenue" FROM sale S, product P
WHERE S.productID = P.prodNo and P.prodName = 'Golf Club' or p.prodName = 'Golf Shoes' ";

The above query is evaluated by the following sequence of steps:

First of all *where* clause is executed and the RIDs of *Product* are selected in which the product's name satisfy the required condition, we get RID p001 and p002 from dimension table *Product* which satisfy this condition. The sum is to be calculated from fact table, so the next step is to find the records from fact table, which are joined with the dimension table having the product id *p001* and *p002*. This is done by using the bitmap join index between the two tables. The bitmap join index for *p001* is:

1	0	1	1	0	1	0	1	0	0	0	0	0	0	1	0	0	0	0	0

and the bitmap join index for *p002* is:

0	1	0	0	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0

The *foundset* or the set of rows from which the final result should be calculated, and these are the rows present in either of these bitmaps. So bitmap (B_f) of the *foundset* can be determined by ORing the above two vectors. The B_f for the above query is following:

1	1	1	1	0	1	0	1	1	0	0	1	0	0	1	0	0	0	0	0

The next step is to calculate the *sum* of *dollar_amt* attribute for those rows which are selected in the *foundset*. For this step, the index on *dollar_amt* can be used and the goal is to have the best index, which can perform this task by scanning the minimum data from the disk. There are three types of indexes that can be used for this purpose. Table 1 presents the cost and performance of using all these indexes in the worst case. Worst case occurs when none of the bitmap has all zeros in it and all of them are to be stored on disk.

Let j is the memory size of data type. i.e., long data type has 64 bits and short has 16 bits. N is the number of tuples in the table.

Table 4.1: Performance of different indexes for aggregates in worst case

	Size on Disk	**Bitmap Scans**	**AND Operations**	**COUNT Operations**	**Total Data Scanned**
Range Encoded Bit-Sliced Index	*J*N* bits	J + 2	J+1	J+2	J*N +2*N bits
Regular Bit-Sliced Index	*J*N* bits	J + 2	J+1	J+1	J*N +2*N bits
Projection Index	*J*N* bits	N/A	N/A	N/A	J*N +2*N bits

From the above table, we can figure that in worst case, all three indexes perform equally. But this is extremely rare when none of the bitmaps have all zeros in them. For a long data type, this is less likely that we have the highest numbers stored in them. Similarly, for a short data type of *dollar_amt*, it is less likely that we have amounts in thousands of dollars, so there is a great chance for highest digits to be zero. Thus we get the bitmaps for higher digits containing all zeros in them. This gives the opportunity of improving the performance. The table 2 below presents the cost and performance of these indexes with the fact table of this example:

Table 4.2: The performance of indexes with the above fact table

	Size on Disk	**Bitmap Scans**	**AND Operations**	**COUNT Operations**	**Total Data Scanned**
Range Encoded Bit-Sliced Index	*10*20* bits	12	11	12	240 bits
Regular Bit-Sliced Index	*10*20* bits	12	11	11	240 bits
Projection Index	*16*20* bits	N/A	N/A	N/A	360 bits

As this is obvious from table 1 and 2 that the first two indexing schemes do not perform worse than the projection index in any case. Thus the projection index is not an option. If we compare the performance of the first two schemes, we can figure out that both are equal in size and scan the same amount of data from disk. They both perform same number of AND operations but first perform one more COUNT operation than second. The count operation is the cheapest among all operations of this processing, so that is highly insignificant. The first index is a range bit-encoded index which also gives good performance for range queries and very straight forward to maintain, while the second one is useful only for aggregates and has to be rebuilt when a data warehouse is updated. Next the performance of different indexes is evaluated for the range queries.

As an other example if the management needs to know that how much revenue was generated through the transactions in which transaction amount was less than $400, the question can be asked in form of following query "*"SELECT sum (S.dollar_amt) "Revnue" FROM sale S WHERE S.dollar_amt <=400 ";*

For the above query, the attribute *dollar_amt* needs to be scanned. Again, this is the goal that the above query should be answered by scanning minimum data from the disk. If a projection index is present, the whole index needs to be scanned and each value has to be compared with the specified amount to figure out the values in the result set, and then those values will be added together. As quick response time is a critical for data warehouse queries, there are some alternatives to projection index which can be used to optimize the response time for such kind of range queries. Following tables present the comparative analysis of all alternatives in space and performance. We will compare the performance of projection index, space optimized range-encoded bit-sliced index proposed in this thesis, uniform base-10 multiple component range-encoded index, and regular bit-sliced index proposed by [OQ97]. Table 3 presents their comparison in worst case and table 4 presents it for this particular case.

q is the number of components in a range-encoded index

Table 4.3: Performance of different indexes for range queries in general

	Size on Disk	**Bitmap Scans**	**Bitmap Operations**	**Total Data Scanned**
Range Encoded Bit-Sliced Index	*j*N* bits	j	j	j*N bits
Regular Bit-Sliced Index	*j*N* bits	j	4*j	j*N bits
Base-10 multiple component	*q*9*N* bits	(q-1)*2+1	(q-1)*2+1	((q-1)*2+1)*N bits
Projection Index	*16*N* bits	N/A	N/A	j*N bits

q is the number of components and its maximum value depends on the data type of the data we are using. For a short data type which cannot have a value greater than 65535, the q can not be more than 5 for base 10 components, since there are maximum five digits in any value. If we make higher base components, the miximum possible value of q will be even less and we will have fewer components and less data will be scnned from the disk.

Table 4.4: Performance of different indexes for range queries with above tables

	Size on Disk	**Bitmap Scans**	**Bitmap Operations**	**Total Data Scanned**	**performance**
Range Encoded Bit-Sliced Index	*10*N* bits	10	10	10*N bits	Reasonable
Regular Bit-Sliced Index	*10*N* bits	10	40	10*N bits	Slowest Not good
Base-10 multiple component	*27*N* bits	5	5	5*N bits	Fastest but 3 times more space
Projection Index	*16*N* bits	N/A	N/A	16*N bits	Slow

As we can see from the above table, higher base multiple components bitmap perform best for range queries but the cost of storing them is high. In this example, the size of 3-component base-10 index is three times greater than the first two indexes. The proposed index takes three times less space and perform better than the other two schemes, at the same time this index performs equal to the best (bit-sliced index) for the aggregates.

4.2 – Performance Analysis from System Implementation

In this section, a performance comparison of space optimized range-encoded index proposed in this thesis with bit-sliced index [OQ97] is introduced with the system implementation of both. The range-encoded bitmap index and bit-sliced index are constructed with various sizes and various ranges of data in the indexed attribute. The programs to construct these indexes for the numeric attribute of a fact table are developed in C. The table ***ACTIVITIES (CustomerID char(4), Acocounttype char(4), Transactiontype varchar2(10), Time char(12), Amount number(8,2))*** represents a banking warehouse database fact table, which has each of its tuple representing a transaction record of bank. The fact table is generated with a PRO*C program. The two algorithms for calculating aggregates and two algorithms for evaluating range with each index are implemented. All four algorithms are implemented in "C". The implementation has two different phases. The first phase is to construct all types of indexes for the numeric attribute of the generated fact table, i.e., transaction amount for a bank's data warehouse with the example fact table.

The experiments are performed on two different platforms. The first set of experiments is conducted on a general-purpose computer system in the university called SGI Challenge XL. The SGI Challenge XL has 16 R4400 processors with 12 processors at 150 MHz, 2 processors at 200 MHz and 2 processors at 250 MHz. The other four sets of experiments are conducted on a stand-alone SUN (Ultra 60) workstation with processor speed of 450 MHz, 512 megabytes of memory and Solaris operating system.

4.2.1 - Experiment 1:

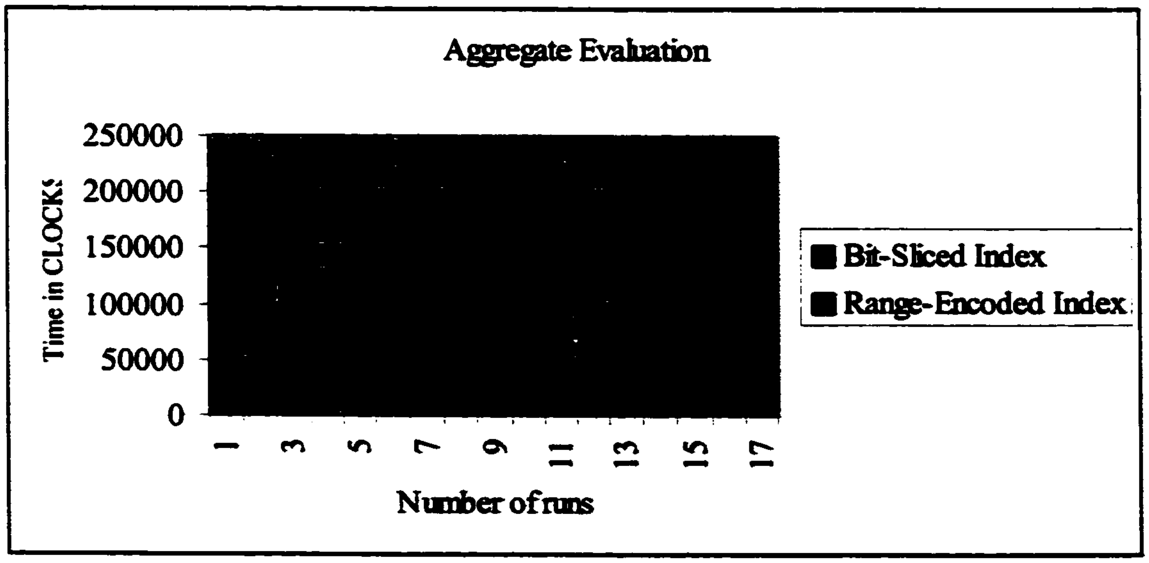
The first set of experiments is carried out by generating a FACT table called ACTIVITIES having 100,000 records in it. The table ACTIVITIES is generated by a PRO*C program. A projection index is built on the *AMOUNT* attribute. The bit-slice index and binary range-encoded bitmap index are built from that projection index.

Aggregate Evaluation: The aggregate queries are evaluated for a variety of different *foundsets* with each index. Each time a different foundset is generated randomly for calculating aggregates, the same foundset is used with both indexes to evaluate their performance. The percentage of the foundset is started from 10% and goes up to 100% with an increment of 10% in each set of experiments. A foundset of 10% means that 1/10th records of fact table are in the result for which we have to add the attribute values. With each percentage of foundset, two to three aggregates are evaluated with both indexes. The time is noted in number of CLOCKS taken by the algorithm to perform the aggregate with each index. CLOCK is the smallest unit used to calculate execution time for a program. The experimental results are shown in table 4.5. The first column of the table stands for the percentage of foundset for the predicate to be evaluated, the second column stands for the number of runs and the third and fourth columns represent the time taken by bit-sliced index (Bs) and range-encoded index (Br) respectively. The fifth column represents the difference in time for both indexes. The positive value in this column represents the time saved with proposed index, and negative value represents that the proposed index took more time and the sixth column represents the percentage gain.

Table 4.5: Performance of both indexes for 27 aggregate queries with foundset 10%-100%

Percentage of Foundset	Number Of Runs	Time (CLOCKS) with Bit-Sliced Index	Time (CLOCKS) with Range-Encoded Index	Difference $B_r - B_s$	Percentage Gain
10%	3	210000	210000	0	0%
20%	1	210000	200000	10000	5%
20%	1	170000	150000	20000	12%
20%	1	130000	130000	0	0%
30%	1	210000	220000	-10000	-5%
30%	2	200000	220000	-20000	-10%
30%	1	170000	150000	20000	12%
30%	1	200000	210000	-10000	-5%
40%	1	160000	150000	10000	6%
40%	2	210000	200000	10000	5%
40%	1	220000	210000	10000	5%
50%	2	130000	120000	10000	8%
60%	1	200000	210000	-10000	-5%
70%	2	200000	210000	-10000	-5%
80%	2	200000	210000	-10000	-5%
90%	2	170000	150000	20000	12%
100%	3	210000	200000	10000	5%

Graph 4.1: Performance of both indexes for 27 aggregate queries of table 4.5



By observing the results, we can see that both indexes perform the same for aggregate queries most of the time. Depending on the foundset, some times the proposed algorithm perform slightly better than the previous algorithm, and some times the previous algorithm performs slightly better than the proposed algorithm. The time unit *CLOCK* is so small that 10,000 is the smallest number reported, and 10,000 *CLOCKS* means a few microseconds. In the above experiment, there was a net gain of 40000 CLOCKS in the time for the proposed index.

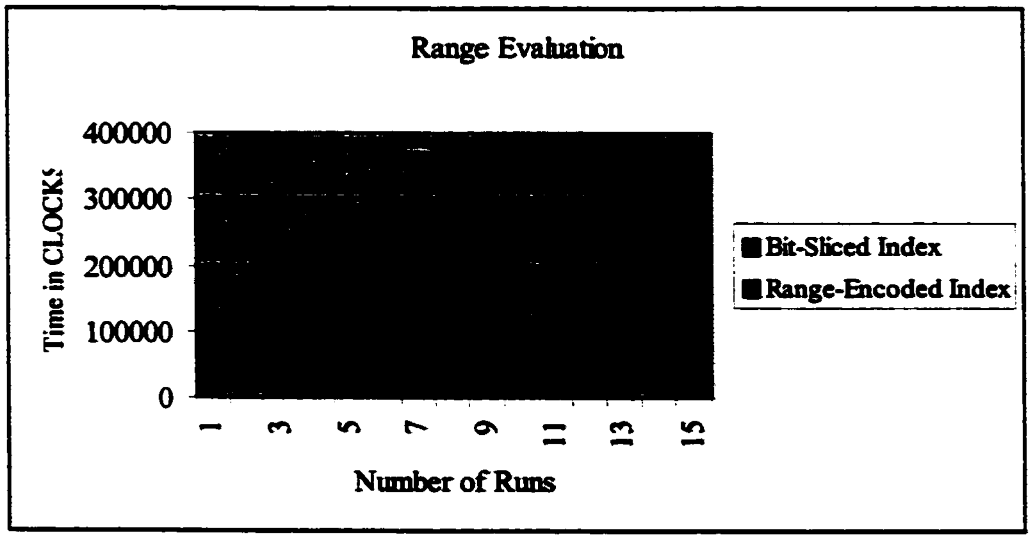
Range Evaluation: Different range predicates are evaluated with both types of indexes. The range was evaluated for a variety of values to gain the general response time for both indexes. The predicate like *AMOUNT* <= *500* is simply represented as (<=500) in the tables below. The first set of range was evaluated for the records having values less or equal to 100 (<= 100), less or equal to 450 (<= 450) and less or equal to 800 (<= 800). Then three queries are evaluated for the records where the values are not equal to 150 (!= 150), not equal to 375 (!= 375), and not equal to 750 (!= 750) respectively. The next set of range queries was evaluated for the records having values greater or equal to 100 (>= 100), records having values greater or equal to 450 (>= 450), and records having values greater or equal to 800 (>=800). The next set of queries is evaluated for the records having values equal to 250 and 650, for the records having values less than 50 and 850 and finally for the records having values greater than 200 and 700.

By observing the experimental results from table 4.6, we can infer that the range-encoded index perform faster than the bit-sliced index. Both indexes scan the same amount of data from storage, but bit-sliced index perform more operations on the data, the larger bitmap means more time difference in their performance. The last column of the table represents the time gained by the range-encoded index. The gain in time ranges from 0 in one case to 130,000 CLOCKS in another.

Table 4.6: Performance of both indexes for different range queries

Predicate Type	Bit-Sliced Index Time in Clocks	Range-Encoded Index Time in Clocks	GAIN in Time	Percentage Gain in Time
<= 100	160000	130000	30000	19%
<= 450	180000	170000	10000	6%
<= 800	320000	250000	70000	22%
!= 150	270000	260000	10000	4%
!= 375	300000	290000	10000	3%
!= 750	310000	290000	20000	6%
>= 100	370000	370000	0	0%
>= 450	300000	290000	10000	3%
>= 800	150000	140000	10000	7%
= 250	210000	120000	90000	43%
= 650	200000	120000	80000	40%
< 50	220000	180000	40000	18%
< 850	360000	230000	130000	36%
> 200	350000	340000	10000	3%
> 700	180000	160000	20000	11%
		Net Gain	**540000**	

Graph 4.2: Performance of both indexes for range queries of table 4.6



4.2.2 - Experiment 2:

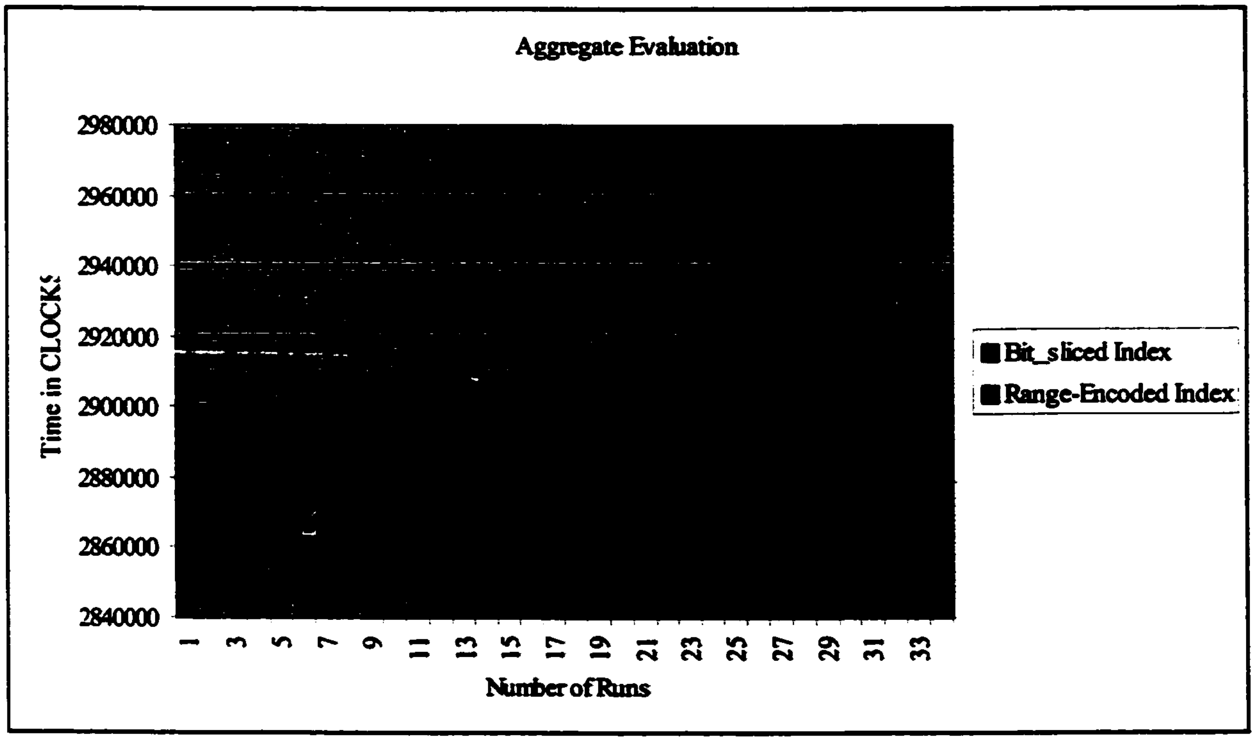
The second set of experiments are carried out by generating the values of AMOUNT attribute randomly. There are one and a half million (1,500,000) values ranging between 0 and 999 (up to three digit numbers). The bit-sliced index and binary range-encoded bitmap index are built from the projection of this attribute. This set of experiments is conducted on a SUN workstation with Solaris operating system. Similar predicates are evaluated for these indexes as have been done in first experiment (section 4.2.1). With each foundset (10% to 100%), five predicates are evaluated in these experiments.

Aggregate Evaluation: There are fifty aggregate queries evaluated for a different foundset each time but same foundset is used with each index. Each time, foundset is generated randomly. The experimental results are shown in table 4.7.

Table 4.7: Performance of both indexes for 50 aggregate queries with foundset 10%-100%

Percentage of Foundset	Number Of Runs	Time (CLOCKS) with Bit-Sliced Index	Time (CLOCKS) with Range-Encoded Index	Difference $B_r - B_s$	Percentage Gain in Time
10%	3	2910000	2920000	-10000	0%
10%	1	2900000	2910000	-10000	0%
10%	1	2920000	2900000	20000	1%
20%	1	2920000	2910000	10000	0%
20%	2	2900000	2920000	-20000	-1%
20%	1	2920000	2910000	10000	0%
20%	1	2910000	2920000	-10000	0%
30%	1	2920000	2910000	10000	0%
30%	2	2900000	2920000	-20000	-1%
30%	1	2910000	2900000	10000	0%
30%	1	2920000	2910000	10000	0%
40%	1	2910000	2910000	0	0%
40%	1	2910000	2920000	-10000	0%
40%	2	2900000	2910000	-10000	0%
40%	1	2910000	2900000	10000	0%
50%	1	2910000	2900000	10000	0%
50%	1	2920000	2900000	20000	1%
50%	1	2900000	2920000	-20000	-1%
50%	1	2900000	2910000	-10000	0%
50%	1	2910000	2910000	0	0%
60%	2	2910000	2910000	0	0%
60%	1	2890000	2910000	-20000	-1%
60%	1	2900000	2920000	-20000	-1%
60%	1	2910000	2900000	10000	0%
70%	1	2950000	2970000	-20000	-1%
70%	3	2950000	2960000	-10000	0%
70%	1	2950000	2950000	0	0%
80%	1	2950000	2960000	-10000	0%
80%	2	2940000	2960000	-20000	-1%
80%	2	2950000	2960000	-10000	0%
90%	2	2910000	2900000	10000	0%
90%	3	2900000	2910000	-10000	0%
100%	3	2900000	2910000	-10000	0%
100%	2	2900000	2900000	0	0%

Graph 4.3: Performance of both indexes for 50 aggregate queries of table 4.7



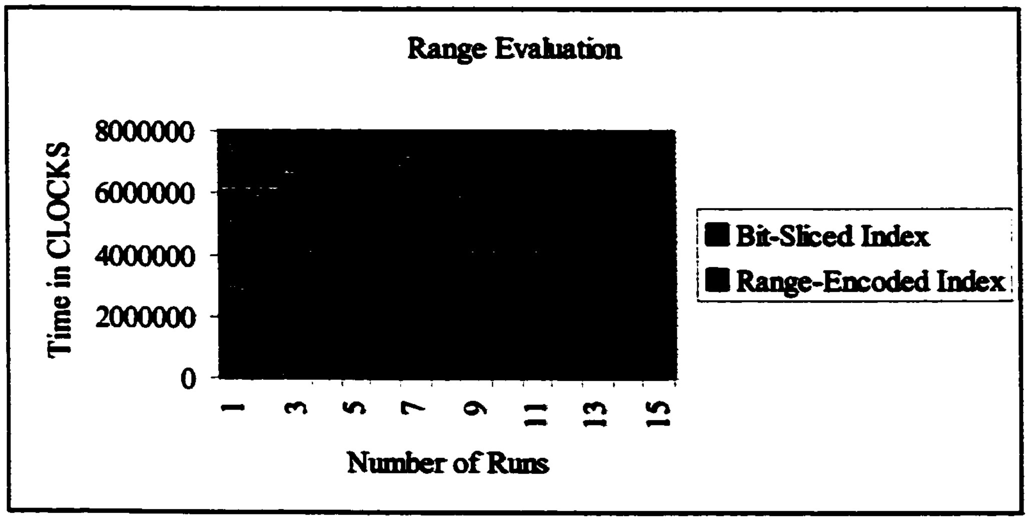
By observing the results, we can figure that both indexes perform the same for seven predicates. Twenty-one times, the proposed index took 10,000 more clocks than bit-sliced index, while ten times, bit-sliced index took 10,000 more clocks than the proposed index. Ten times, the proposed index took significantly longer *(>=20,000 Clocks)* than the bit-sliced index and two times the bit-sliced index took significantly longer than the proposed index but that significant difference is not more than a few microseconds. As a general rule, their performance will be considered equal.

Range Evaluation: Range predicates of the same types are evaluated with both types of indexes as in the first experiment (section 4.2.1). The first set of range predicates was evaluated for the records having values less or equal to 100 (<= 100), less or equal to 450 (<= 450) and less or equal to 800 (<= 800). Then, three queries are evaluated for the records where the values are not equal to 150 (!= 150), not equal to 375 (!= 375), and not equal to 750 (!= 750) respectively. The next set of range queries is evaluated for the records having values greater or equal to 100 (>= 100), records having values greater or equal to 450 (>= 450), and records having values greater or equal to 800 (>=800). The next set of queries is evaluated for the records having values equal to 250 and 650, for the records having values less than 50 and 850 and finally for the records having values greater than 200 and 700.

Table 4.8: Performance of both indexes for different range queries

Predicate Type	Bit-Sliced Index Time in Clocks	Range-Encoded Index Time in Clocks	GAIN in Time	Percentage Gain in Time
<= 100	2830000	2770000	60000	2%
<= 450	4690000	4580000	110000	2%
<= 800	6550000	6360000	190000	3%
!= 150	7590000	7330000	260000	3%
!= 375	7560000	7360000	200000	3%
!= 750	7580000	7370000	210000	3%
>= 100	7050000	6850000	200000	3%
>= 450	5200000	5070000	130000	3%
>= 800	3370000	3290000	80000	2%
= 250	2310000	2270000	40000	2%
= 650	2300000	2270000	30000	1%
< 50	2560000	2520000	40000	2%
< 850	6760000	6590000	170000	3%
> 200	6530000	6330000	200000	3%
> 700	3880000	3800000	80000	2%
		Net Gain	**2000000**	

Graph 4.4: Performance of both indexes for range queries of table 4.8



By examining the experimental results from table 4.8 and graph 4.4, we can observe that the range-encoded index performs much faster than the bit-sliced index for larger tables. The time gain ranges from 30,000 clocks to 260,000 clocks. The performance gap widens because the indexes are larger and each operation on them takes a little longer than the previous experiment.

Next two experiments are carried out to confirm these results with different data and different table sizes. Both of the following experiments are conducted on same SUN Solaris machine of experiment two.

4.2.3 - Experiment 3:

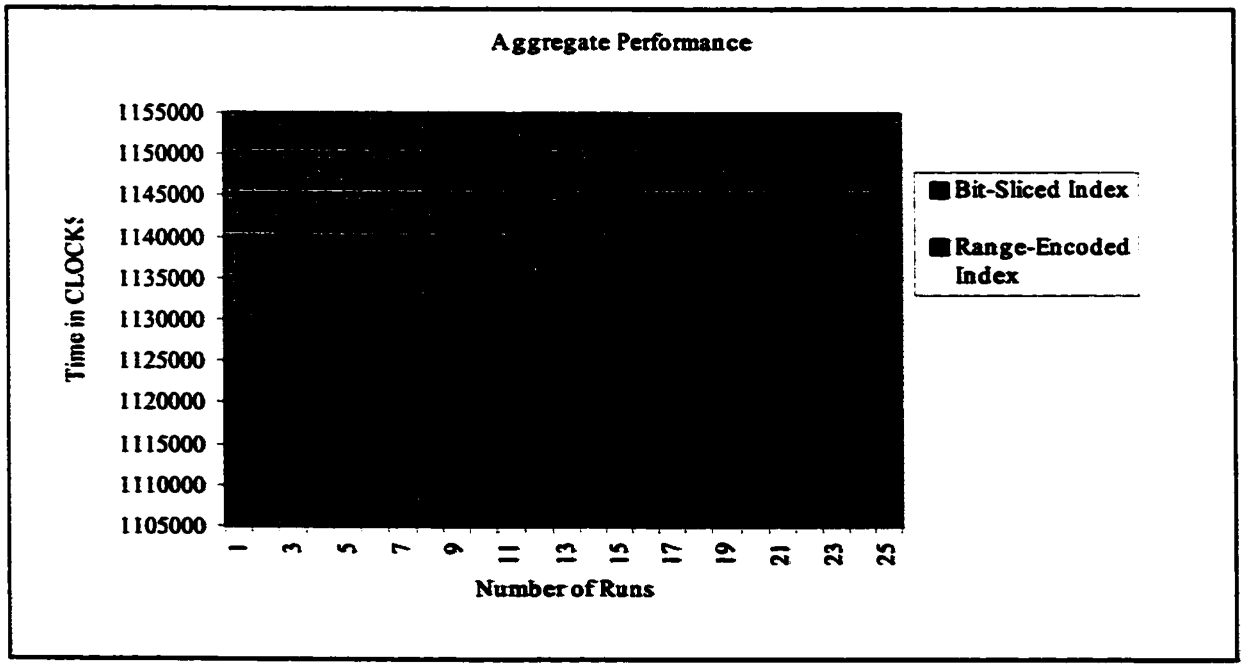
The third set of experiments carried out by randomly generating the values of *AMOUNT* attribute between the range of 0 – 9999 (up to four digit numbers). There are four hundred thousand (400,000) records in the table. The bit-sliced index and binary range-encoded bitmap index are built from the projection of this attribute and similar sets of experiment are carried out again with different predicate values. Since the attribute values are larger, more bitmaps are needed to store this data. In fact, each index has 14 bitmaps in them.

Aggregate Evaluation: There are forty aggregate queries evaluated by generating a different foundset each time. The same foundset is used with both indexes to evaluate their performance. The experimental results are shown in table 4.9.

Table 4.9: Performance of both indexes for 50 aggregate queries foundset (10% - 100%)

Percentage of Foundset	Number Of Runs	Time (CLOCKS) with Bit-Sliced Index	Time (CLOCKS) with Range-Encoded Index	Difference $B_r - B_s$	Percentage Gain in Time
10%	2	1130000	1140000	-10000	-1%
10%	3	1140000	1140000	0	0%
10%	1	1140000	1150000	-10000	-1%
20%	3	1130000	1140000	-10000	-1%
20%	2	1140000	1140000	0	0%
30%	3	1130000	1140000	-10000	-1%
30%	2	1140000	1140000	0	0%
40%	1	1130000	1140000	-10000	-1%
40%	1	1140000	1140000	0	0%
40%	1	1140000	1150000	-10000	-1%
40%	2	1130000	1150000	-20000	-2%
50%	2	1130000	1150000	-20000	-2%
50%	2	1130000	1140000	-10000	-1%
50%	1	1140000	1140000	0	0%
60%	2	1130000	1150000	-20000	-2%
60%	2	1140000	1140000	0	0%
60%	1	1130000	1130000	0	0%
70%	3	1130000	1140000	-10000	-1%
70%	1	1120000	1140000	-20000	-2%
70%	1	1140000	1140000	0	0%
80%	4	1130000	1140000	-10000	-1%
80%	1	1130000	1150000	-20000	-2%
90%	1	1140000	1140000	0	0%
90%	4	1130000	1140000	-10000	-1%
100%	5	1130000	1140000	-10000	-1%

Graph 4.5: Performance of both indexes for 50 aggregate queries of table 4.9



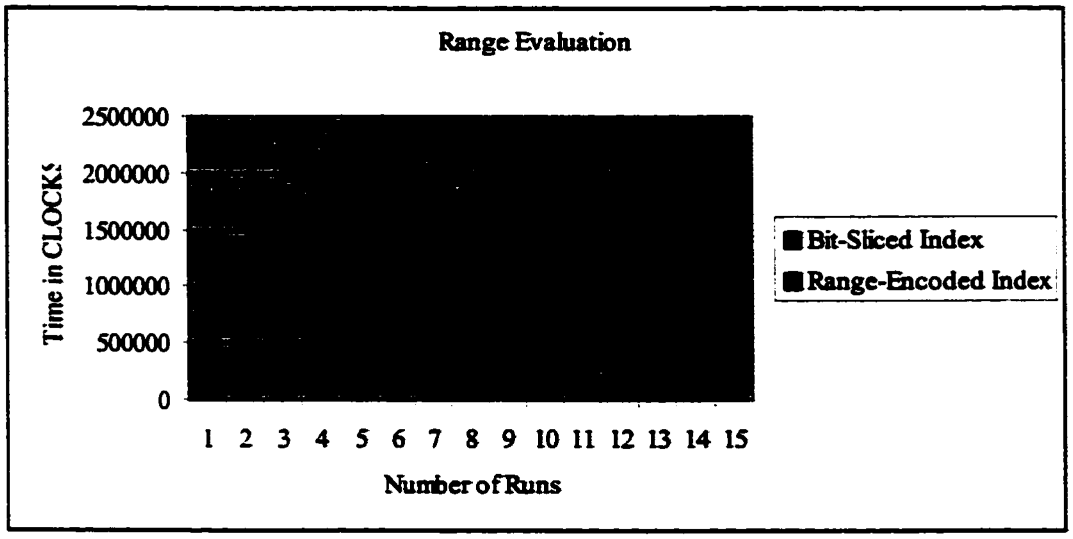
By observing the results of table 4.9, we can figure that both indexes perform exactly the same for *fourteen* queries. About *twenty-eight* times, there was an insignificant difference in their performance and for *eight* times bit-sliced index took 20,000 less CLOCKS than the range-encoded index.

Range Evaluation: Following the same pattern, range predicates are evaluated with both types of indexes. The first set of range was evaluated for the records having values less or equal to 1000 (<= 1000), less or equal to 4500 (<= 4500) and less or equal to 8000 (<= 8000). Then three queries are evaluated for the records where the values are not equal to 400 (!= 400), not equal to 3750 (!= 3750), and not equal to 7500 (!= 7500) respectively. The next set of range queries were evaluated for the records having values greater or equal to 1000 (>= 1000), records having values greater or equal to 4500 (>= 4500), and records having values greater or equal to 8000 (>= 8000). The next set of queries is evaluated for the records having values equal to 2500 and 65000. For the records having values less than 500 and 8500 and finally for the records having values greater than 200 and 7000.

Table 4.10: Performance of both indexes for different range queries

Predicate Type	Bit-Sliced Index Time in Clocks	Range-Encoded Index Time in Clocks	GAIN in Time	Percentage Gain in Time
<= 1000	960000	950000	10000	1%
<= 4500	1430000	1400000	30000	2%
<= 7500	1920000	1880000	40000	2%
!= 400	2190000	2150000	40000	2%
!=3750	2190000	2140000	50000	2%
!= 7500	2220000	2140000	80000	4%
>= 1000	2060000	2010000	50000	2%
>= 4500	1580000	1540000	40000	3%
>= 8000	1110000	1080000	30000	3%
= 2500	810000	810000	0	0%
= 6500	820000	810000	10000	1%
< 500	880000	880000	0	0%
< 8500	1980000	1940000	40000	2%
> 200	2170000	2120000	50000	2%
> 7000	1230000	1210000	20000	2%
		Net Gain	**490000**	

Graph 4.6: Performance of both indexes for range queries of table 4.10



By observing the experimental results from table 4.10, we can again infer that the range-encoded index performed faster than the bit-sliced index. The gain in time ranges from 0 to 80,000 CLOCKS.

4.2.4 - Experiment 4:

The fourth set of experiments was carried out by randomly generating the values of *AMOUNT* attribute between the range of 0 – 65535 (up to five digit numbers). There are three hundred thousands (300,000) records. Bit-sliced index and binary range-encoded bitmap index are built the same way and stored in their respective files as done previously.

Aggregate Evaluation: Again, there are forty aggregate queries evaluated by generating a different foundset each time and using the same foundset with both indexes to evaluate their performance. The percentage of the foundset is not controlled this time, each found set happened to be generated between 45% to 55%. Since the data values are even higher in these experiments, so there are more bitmaps in the index. The experimental results are shown in table 4.11. The graphs of following two tables are not included, since the result can easily be found from the last column of the table.

Table 4.11: Performance of both indexes for 40 aggregate queries

No. of Runs	Bit-Sliced Index Time in Clocks	Range-Encoded Index Time in Clocks	Difference in Time (Br - Bs)
19	980000	980000	0
2	970000	970000	0
3	990000	970000	20000
3	980000	970000	10000
10	970000	980000	-10000
1	980000	990000	-10000
2	970000	990000	-20000

From table 4.11, we can observe that, both indexes perform same for twenty-one times, fourteen times, one performed slightly better than other. Three times, the proposed index performed significantly better than bit-sliced index and two times, bit-sliced index performed better than the proposed range-encoded index.

Range Evaluation: Similar range predicates are evaluated with both types of indexes. The first set of ranges were evaluated for the records having values less or equal to 1000 (<= 1000), less or equal to 30,000 (<= 30,000) and less or equal to 55000 (<= 55000). Then three queries were evaluated for the records where the values are not equal to 20,000 (!= 20,000), not equal to 37500 (!= 37500), and not equal to 63,000 (!= 63,000) respectively. The next set of range queries were evaluated for the records having values greater or equal to 1000 (>= 1000), records having values greater or equal to 25000 (>= 25000), and records having values greater or equal to 58000 (>= 58000). The next set of queries is evaluated for the records having values equal to 2500 and 65000, for the records having values less than 5000 and 55000 and finally for the records having values greater than 2000 and 57000. Table 4.12 presents the results of range queries with the predicates provided in the first column.

Table 4.12: Performance of both indexes for different range queries

Predicate Type	Bit-Sliced Index Time in Clocks	Range-Encoded Index Time in Clocks	GAIN in Time
<= 1000	720000	700000	20000
<= 30000	1170000	1150000	20000
<= 55000	1570000	1520000	50000
!= 20000	1730000	1680000	50000
!= 37500	1740000	1680000	60000
!= 63000	1720000	1680000	40000
>= 1000	1720000	1690000	30000
>= 25000	1330000	1300000	30000
>= 58000	820000	820000	0
= 2500	700000	680000	20000
= 65000	690000	690000	0
< 5000	770000	760000	10000
< 55000	1560000	1520000	40000
> 2000	1690000	1640000	50000
>57000	820000	820000	0

From table 4.12, it can easily be observed that the bit-sliced index is out performed by 0 to 60,000 CLOCKS in each query.

4.2.5 - Experiment 5:

The last set of experiments was carried out by randomly generating the values of *AMOUNT* attribute between the range of 0 – 999 (up to three digit numbers). There are 1.9 million records. Bit-sliced index and range-encoded bitmap index were constructed similarly and were stored in their respective files as have been done previously.

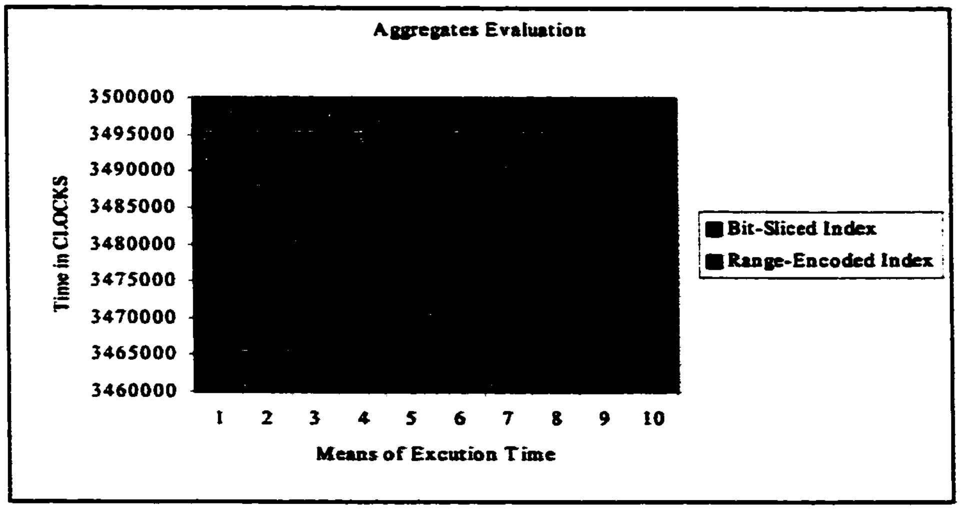
Aggregates: The foundset was generated randomly and its percentage was varied from 10% to 100%. With each percentage, twenty aggregates are evaluated with both indexes using the same foundset. The mean of execution time is calculated for each percentage of foundset. For each index, the standard deviation of their mean is calculated and test of significance (T-test) is performed for the difference in their mean time of execution. The T-test is a test which evaluates the significance of a hypothesis. In this case, the hypothesis was made that the mean time of execution with both indexes is same. The T-test was performed on the hypothesis that the mean time of both indexes were same with 95% confidence. The results of the T-test are presented in table 4.13. The mean execution time for bit-sliced index is 3482500 *CLOCKS*, and for range-encoded index is 3490450 *CLOCKS*.

Table 4.13: T-test results for aggregate evaluation

t-Test: Paired Two Sample for Means		
	Bit-Sliced Index	***Range-Encoded Index***
Mean	3482500	3490450
Variance	61055276.38	62610552.76
Standard Deviation	7813.787582	7912.683032
Observations	200	200
Pearson Correlation	0.371837043	
Hypothesized Mean Difference	0	
df	199	
t Stat	-12.75589521	
P(T<=t) one-tail	6.31645E-28	
t Critical one-tail	1.652547326	
P(T<=t) two-tail	1.26329E-27	
t Critical two-tail	1.971957317	

By observing the results of T-test for aggregates, we can see that the *t* statistic is –12.755, which is less than *t* critical value of 1.971, so we cannot reject the hypothesis. In other words, we can say with 95% confidence that the mean time taken by both indexes for calculating aggregates is equal.

Graph 4.7: Performance of both indexes with their mean time for aggregates

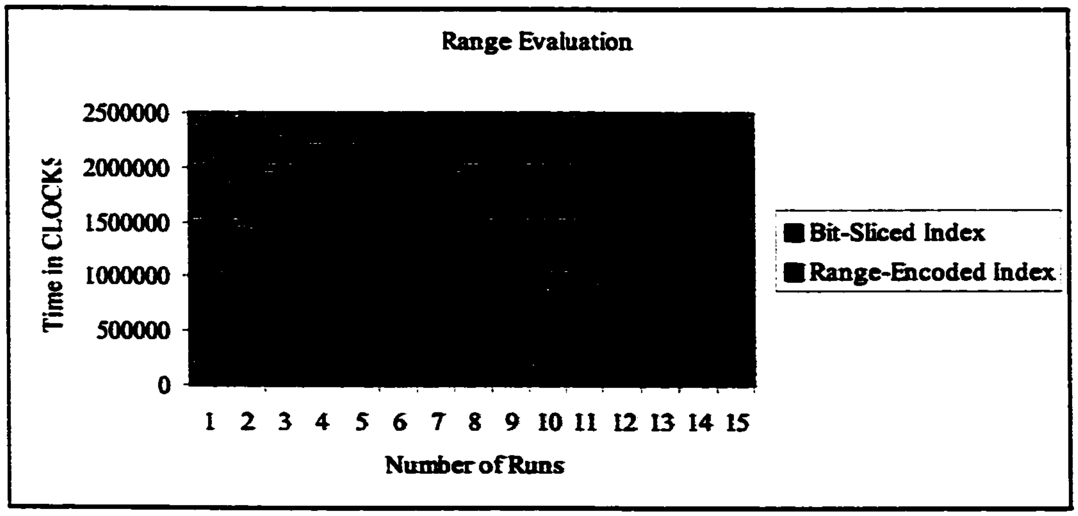


Range: The range predicates were evaluated with both indexes. There were fifteen evaluations with each type of range predicate. The mean execution time was calculated for both indexes for their performance with range queries. The T-test was performed on the hypothesis that with 95% confidence, it can be claimed that the mean time of both indexes is same for evaluating range. The mean execution time with bit-sliced index was 5844400 *CLOCKS*, and with range was 4984000 *CLOCKS*. The results of T-test are presented in table 4.14. By observing those results we can see that the t statistic is 21.111 and t critical value is 1.986, so we conclude that the hypothesis should be rejected. In other words by looking at the difference of those values, we claim with 95% confidence that the mean execution time for range-encoded index was far less than the bit-sliced index.

Table 4.14: T-test results for range evaluation

t-Test: Paired Two Sample for Means		
	Bit-Sliced Index	***Range-Encoded Index***
Mean	5344000	4984000
Variance	3.9382E+12	3.82619E+12
Standard Deviation	1984488.613	1956065.071
Observations	90	90
Pearson Correlation	0.996733093	
Hypothesized Mean Difference	0	
df	89	
t Stat	21.111319	
P(T<=t) one-tail	1.02559E-36	
t Critical one-tail	1.662156137	
P(T<=t) two-tail	2.05119E-36	
t Critical two-tail	1.986977622	

Graph 4.8: Performance of both indexes with mean time for range queries



5. CONCLUSIONS AND FUTURE DIRECTIONS

A space optimized range-encoded bitmap index is proposed to replace the bit-sliced index. The proposed algorithm gives at least similar performance for the aggregate evaluation with the advantages of easy maintenance and the possibility of using it for range predicates more efficiently. The option of multiple indexes is still there for better query performance for range queries with an index of higher base components. There are two options for storing the bitmap of all 1's. First, it can be stored and compressed by the data compression techniques, second, only the information like the size of a bitmap of all 1's can be stored and can be generated in memory whenever needed. From the evaluation of both indexing techniques, it is obvious that the proposed index gives similar performance for aggregates and performs considerably better for the range queries. The experimental evaluation is done with a variety of values and predicates. The execution is taken in number of CLOCKS, which is the smallest time unit returned by the computer. The reason behind using the CLOCKS is simply to emphasize the difference in performance. The performance gap between the two widens for range evaluation when the underlying data set is huge while the performance for aggregates remains the same.

The use of these algorithms can be generalized for different data types. We have restricted ourselves to the use of *short* data type and the attribute values of integer type. However, the attribute values are necessarily fractional values. Our proposed scheme can easily be adapted for other data types. In the experimental analysis, we restricted ourselves to *short* data type, which has 16 bits in it, resulting in 16 bitmaps maximum for each index. This restriction can easily be removed with the use of *long* or other data types.

Future researchers should look into the development of efficient algorithms for evaluation of aggregates with higher based (base 3 or more) multiple components bitmap index.

REFERENCES

[ASK00] Jung-Ho Ahn, Ha-Joo Song and Hyoung-Joo Kim, **Index set: A practical indexing scheme for object database systems,** *Elsevier journal of Data and Knowledge Engineering, Vol. 33, No. 3, pages 199-217, 2000*

[A-YJ00]. S.A. Amer-Yahia, T. Johnson. **Optimizing Queries On Compressed Bitmaps,** *Proceedings of the 26th VLDB Conference, 2000, Pages 329 - 338.*

[BBFJ00] A. Benander, B. Benander, A. Fadlalla, G. James, **Data warehouse administration and management** , *Information Systems Management, Vol. 17, No. 1, pages 71 - 80, 2000*

[BOS98]. **Indexing techniques for advanced database systems,** *Book by E. Bertino, B.C. OOI, R. Sacks-davis et. al. 1998, ISBN 0-7923-9985.*

[BS97]. Alex Berson, S. J. Smith. **Data Warehousing, Data Mining, and OLAP,** *McGraw-Hill, 1997.*

[CD97]. S. Chaudhri and U. Dayal. **An Overview of Data Warehousing and OLAP Technology**. *ACM SIGMOD Record, 26(1): 65-74, 1997.*

[CI97] C.-Y. Chan, Y.E. Ioannidis, **Bitmap Index Design and Evaluation,** *CS Dept., University of Wisconsin-Madison, hhtp://www.cs.wisc.edu/~cydhan/paper101.ps, 1997.*

[CI98]. C-Y. Chan and Y. E. Ioannidis. **Bitmap Index Design and Evaluation,** . *In Proc. of the ACM SIGMOD Conference on Management of Data, 1998, pages 355-366.*

[CI99]. Chee Yong Chan, Yannis E. Ioannidis: **An Efficient Bitmap Encoding Scheme for Selection Queries.** *In Proc. of the ACM SIGMOD Conference on Management of Data, 1999, pages 215-226*

[Co93]. E. F. Codd. **Providing OLAP: An IT Mandate Unpublished Manuscript,** *E.F. Codd and Associates, 1993.*

[Ed95]. H. Edelstein, Technology Analysis: **Faster Data Warehouses**, Article published in Information Week, Dec 4, 1995. http://www.informationweek.com/556/56olbit.htm

[Ez01]. C.I. Ezeife. **Selecting and Materializing Horizontally Partitioned Warehouse Views,** *Elsevier journal of Data and Knowledge Engineering, Vol. 36, No. 2, pages 185-210, 2001*

[GHRU96]. H. Gupta, V. Harinruayen, A. Rajaraman and J. D. Uliman. **Index Selection for OLAP**, *Proceedings of the Intl. Conference on Data Engineering (ICDE) , 1996, pages 208 - 219.*

[GBLP96]. J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. **Data cube: A relational aggregation operator generalizing group-by, cross-tabs and subtotals.** *In Proceedings of the 12th Int'l Conference on Data Engineering 1996, pages 152-159.*

[GDCG91]. E.L. Glaser, P. DesJardins, D. Caldwell, and E.D. Glaser. **Bit string compressor with Boolean operation processing capability**. July 1991. U.S. Patent # 5036457.

[GHQ95]. A. Gupta, V. Harinarayan, Dallan Quass. **Aggregate-Query Processing in Data Warehousing Environments,** *Proceedings of the 21st International Conference on Very Large Databases, Zurich, Switzerland, 1995, pages 358 - 369.*

[Gr93]. G. Graefe. **Query Evaluation Techniques for Large Databases.** *ACM Computing Surveys, 25(2):73-170, 1993.*

[GRDTV98] . K. Goyual, K. Ramamritham, A. Datta and H. Thomas, I. Viguier. **Have your data and index it too, An effiicent approach for data warehouse storage.** *Technical Report, University of Arizona, Tucson, 1998.*

[GRDT99]. K. Goyual, K. Ramamritham, A. Datta and H. Thomas. **Indexing and Compression in Data Warehouses,** *Technical Report, University of Arizona, Tucson, 1999.*

[HAMS97]. C-T. Ho, R. Agrawal, N. Megiddo, R. Srikant. **Range Queries in OLAP Data Cubes.** *In Proc. of the ACM SIGMOD Conference on Management of Data, 1997, pages 73 - 88.*

[HRU96]. V. Harinarayan, A. Rajaraman, J.D. Ullman. **Implementing Data Cubes Efficiently.** *In Proc. of the ACM SIGMOD Conference on Management of Data, 1996, pages 205 – 216*

[JKS00]. H. V. Jagadish, N. Koudas, D. Srevastava. **On Effective Multi-Dimensional Indexing for Strings.** *Proc. ACM SIGMOD Intl. Conference on Management of Data, Dallas, USA, 2000, pages 403 - 407.*

[JL99]. M. Jurgens,H.-J. Lenz, **Tree Based Indexes vs. Bitmap Indexes: A Performance Study,** *Proceedings of the International Workshop on Design and Management of Data Warehouses (DMDW'99)*

[Jo99]. T. Johnson. **Performance Measurements of Compressed Bitmap Indexes.** *In Proc. Conf. Very Large Data Bases, 1999.*

[K99]. M. Kornacker. **High-Performance Extensible Indexing.** *Proceedings of the 25th VLDB Conference, Edinburgh, Scotland, 1999.*

[KR97] Y. Kotidis , N. Roussopoulos. **A Generalized Framework for Indexing OLAP Aggregates.** *Technical Report CS-TR-3841, University of Maryland, Oct. 1997.*

[LQB97]. W.J. Labio, D. Quass, B. Adelberg. **Physical Database Design for Data Warehouses,** *International Conference on Data Engineering (ICDE) 1997. Pages 277 - 288*

[LS90]. David Lomet and Betty Salzberg. **The Hb-Tree: a multi-attribute indexing method with good guaranteed performance.** *ACM TODS, 15(4): pages 625 - 658, December 1990.*

[LWMO00] Weifa Liang, Hui Wang and Maria E. Orlowska , **Range queries in dynamic OLAP data cubes,** *Elsevier journal of Data and Knowledge Engineering, Vol. 34, No. 1, pages 21-38, 2000*

[On87]. P. O'Neil. **Model 204 Architecture and Performance.** *2nd Intl. Workshop on High Performance Transaction Systems, Sept, 1987, Pages 40 - 59.*

[OG95]. P. O'Neil and G. Graefe. **Multi-table joins through bitmapped join indexes.** *ACM SIGMOD Record, 24: 1995, pages 8-11.*

[OQ97]. P. O'Neil and D. Quass. **Improved query performance with variant indexes.** *Proc. ACM SIGMOD Intl. Conference on Management of Data,1997, Pages 38-49.*

[ROO01] D. Rinfret, P. O'Neil, E. O'Neil, **Bit-Sliced Index Arithmetic,** *Proceedings of the ACM SIGMOD Conference, Pages 171 - 180, 2001*

[Sa97]. S. Sarawagi. **Indexing OLAP Data,** *Bulletin of the Technical Committee on Data Eng., Vol. 20, March, 1997*

[SR96]. B. Salzberg and A. Reuter. **Indexing for Aggregation,** *Working paper 1996, http://www.ccs.neu.edu/home/salzberg/proj2001.html*

[TPC96]. **Transaction Processing Performance Council (TPC),** *TPC Benchmark D, Decision Support, Standard Specification Revision 1.2.1, Dec 15, 1996.*

[WB97]. M-C. Wu and A.P. Buchmann. **Encoded Bitmap Indexing for Data Warehouses,** *Tech. Report DVS97-3, CS Dept., Technische Universitat Darmstadt,* (http://www.informatik.tudarmstadt.de/DVS1/staff/wu.germam.html), *July 1997.*

[WB97]. M.-C. Wu, A. P. Buchmann. **Research Issues in Data Warehousing,** *Datenbanksysteme in Buro, Technik und Wissenschaft, Springer Verlag 1997.*

[WB98]. M-C. Wu and A.P. Buchmann. **Encoded Bitmap Indexing for Data Warehouses,** *In Proceedings of the 14th Int'l Conference on Data Engineering 1998, pages 220-230.*

[Wu99]. M.-C. Wu. **Query Optimization for Selections Using Bitmaps.** *In Proc. of the ACM SIGMOD Conference on Management of Data, 1999, pages 227-238.*

[WY96]. K.L. Wu, P.S. Yu., **Range-Based Bitmap Indexing for High Cardinality Attributes with Skew,** *Research Report, IBM Watson Research Center, May 1996.*

[ZDN97]. Y. Zhao, P. M. Deshpande, J. F. Naughton. **An Array-Based Algorithm for Simultaneous Multidimensional Aggregates,** *Proc. ACM SIGMOD Intl. Conference on Management of Data, AZ, USA, 1997, pages 159 - 169.*

VITA AUCTORIS

Kashif Bhutta was born in 1967 in Lahore, Pakistan. He graduated from Central Model high school in 1982. He finished his first bachelor's in 1988 with mathematics and physics from Punjab University, Lahore. In 1998, he finished his Bachelors of Commerce from McGill University, Montreal, Canada with major in management information systems and minor in computer science. He is currently a candidate for the Master's degree in school of Computer Science at the University of Windsor and will graduate in the winter term, 2002.