MODIFIED B+ TREE TECHNIQUE IN THE DATA WAREHOUSE ENVIRONMENT

By

ZHENGRONG YI

Bachelor of Science

Habin Industrial University

Habin, P.R.China

1991

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
In partial fulfillment of
The requirements for
The Degree of
MASTER OF SCIENCE
December, 2000

MODIFIED B+ TREE TECHNIQUE IN THE DATA
WAREHOUSE ENVIRONMENT

Thesis Approved:

[illegible]
Thesis Advisor

[illegible]

J Chandler

[illegible]
Dean of the Graduate College

ACKNOWLEDGEMENTS

I sincerely thank my thesis advisor, Dr. J. Terry Nutter, for her intelligent supervision, constructive guidance, warm encouragement, and valuable time she has given me toward the completion of my thesis. My sincere appreciation extends to Dr. J. Chandler and Dr. Jing Peng for serving on my committee; their guidance, encouragement, assistance, and friendship are invaluable.

I would like to give my special thanks to my wife, Liying Zhang, for her love, encouragement, patience, and understanding throughout my study at Oklahoma State University. My respectful thanks go to my parents for their love and encouragement.

Finally, I would like to thank all the faculty of the Department of Computer Science for their support during my studying here.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract

Fast query response time is one of the most important measures for the data warehouse (DWH) environment. Indexing is one key to achieving this objective. Indexing techniques in the relational database area have existed for decades. Many good techniques have been developed, including B trees, hash functions, and others. But the differences between the DWH environment and traditional relational database systems require new or modified techniques because the existing indexing techniques are inadequate for online analytical processing (OLAP) or decision support system (DSS) applications. This thesis presents a modified B+ tree technique in order to support OLAP applications. The proposed technique has the advantages of both B+ tree and bitmap techniques and solves the space problem of the simple bitmap index and the cooperation problem of traditional B+ trees. Performance of the proposed indexing technique is compared to that of the traditional B+ trees by simulations based on the APB-1 benchmark

Chapter 1

Introduction

Online Transaction Processing and Online Analytical Processing

In the past, the primary focus of database systems has been online transaction processing (OLTP). However, modern environments call for sophisticated systems that serve online analytical processing (OLAP) to support management needs in the decision-making process, known as decision support systems (DSS) processing. The purpose of OLAP is to turn large amounts of data into valuable, accessible business information. OLAP and OLTP applications have sufficiently different characteristics that modern systems typically separate OLTP completely from OLAP. Current database technology has evolved to satisfy the requirements of OLTP, but not OLAP systems. Data that support OLAP have to be stored in very large, specialized repositories (over 100GB databases), called data warehouses.

Data Warehouse

A data warehouse (DWH) is a structure that contains the data and process managers that make large-scale information available so that users can make better and faster decisions. It brings large volumes of data obtained from OLTP together with legacy operational systems. Typically, the data warehouse is maintained separately from the organization's operational databases. Characteristics of the data in a DWH include:

- large volume (billions of records),
- effectively static nature (no update, insert, or delete operations -- only append operation needed), and

- historical data with time parameters.

There are two distinct categories of DWH [CD96]:

1. Multidimensional OLAP (MOLAP) server-based implementations, and
2. Relational OLAP (ROLAP) server-based implementations.

Both have advantages and disadvantages, a discussion of the tradeoffs can be found in on [CD96].

The data in a warehouse is typically modeled multidimensionally. Complex and iterative queries such as "What are the ten fastest-growing products this year versus last year, and what are their contribution to total sales?" are very common in DSS. Complex queries of this kind could take several hours or days to process because of the immense volume of data involved. Therefore, one critical aspect of DSS involves improving the speed of processing these queries.

Index Structure

Index structures allow fast random access to records in a file. An index works in much the same way as a catalog in a library. That is, if we are looking for a book on a particular subject, we search the subject catalog alphabetically to locate the subject's entries without checking every card to find the one we want. Database systems use many indexing methods. Each improves query processing for target sets of situations or information.

For the most part, OLAP queries are grouped by varying combinations of columns known as dimensions. These groupings of columns form data cube queries [GBLP96]. The dimensional combinations that define data cube queries are known in advance, so that sets of summary tables can be created to support evaluating an expected set of queries efficiently [AAD+96, HRU96]. However, DSS also involves another kind of OLAP queries, which cannot be identified in advance. Instead, systems must filter rows dynamically using selection criteria that are not pre-computed in dimension tables. Hence it is important to develop sophisticated, complex indexing methods to provide adequate performance not only for pre-planned queries but also for unplanned queries through inexpensive index updating. The objective is fast response to both anticipated and novel queries. Other techniques can also improve query processing in data warehousing. These techniques include optimization techniques, memory management, pre-computation of summarized data, and predefined access paths. All of these can provide important benefits, both alone and in combination with each other. The focus of the current work targets indexing methods for improved Relational OLAP.

Organization of the thesis

Chapter 2 discusses existing techniques and analyzes the use of simple bitmap and standard B+ tree indexes in DWH environments. Chapter 3 presents a new modification of B+ tree indexing that shares the advantages of bitmap approaches. In Chapter 4, that new approach is evaluated through simulations of a DWH environment and application of performance measures. Finally Chapter 5 presents the conclusions of the current work, and indicates new lines of research that it opens.

Chapter 2.

Existing Techniques

- Value-list indexes
- Projection indexes
- Bit-Sliced indexes
- Join indexes

Value-List Indexes

Value-list indexes are traditionally implemented in one of two ways. The B+ tree implementation has been popularly used by many RDBMS. By contrast, the bitmap approach has often been used in DWHs since it dramatically reduces disk I/O by using Boolean operations performed in memory.

B-tree Indexes

B-tree indexes reference each row individually using a *Row Identifier (RID)* that specifies the row's disk position. Each distinct key value entry in the B-tree contains a sequence of RIDs known as an RID-list. When this technique is used to index attributes with a small range of discrete values (e.g., gender), most values are associated with many RIDs, and the efficiency of the B-tree suffers. This deficiency becomes even more critical in data warehouse systems, making bitmap indexes more suitable in this situation.

Bitmap Indexes [OD97]

By contrast to the use of RID-lists in B-trees, bitmap indexes associate key values with bitmaps, in which each bit corresponds to a possible RID. A mapping function converts the bit position to an actual RID. A bitmap for value *v* of attribute *A* is a *bitmap vector*, each of whose bit positions corresponds to a row of the table *T*. The bit is *on* (has the value 1) just in case the corresponding row has value *v* for attribute *A;* otherwise the bit is off (has value 0). Hence, bitmap indexes provide the same functionality as regular indexes, but do so using a different, more efficient internal representation. If the number of distinct key values is small, bitmaps are very space efficient. This indexing technique was first introduced in model 204 [One87].

Bitmap indexes are an alternative for representing an RID-list in a Value-List index. They are simple to represent, use space more efficiently than RID-lists when the number of different key values for the index is small, and are more CPU-efficient. Bitmap indexing improves complex query performance because of the speed of bit-wise operations such as ANDs, ORs, and NOTs applied to bitmap index columns.

Encoded Bitmaps [WB97] modify the standard bitmap index approach to provide greater storage efficiency and better performance for range searches under certain circumstances. However, the modified technique introduces trade-offs and uncertainty for encoded bitmap indexes. One such trade-off involves the overhead of encoding and of searching the mapping table. In addition, bitmap encoding can be difficult to maintain when an attribute field undergoes domain expansion.

Projection Index [OD97]

The main purpose of projection indexes is to reduce the cost of querying a particular attribute field. The basic idea of a projection index on a column *C* is as follows. For a column *C* of a table *T*, all column values are duplicated. This index is comparable to vertical partitioning. Projection indexes work faster than other techniques when only the column values are desired as opposed to the table rows themselves, because the actual tuples of the fact table need not be accessed at all. Projection indexes are implemented in Sybase IQ.

Bit-Sliced Indexes [OD97]

Bit-Sliced Indexes are based on the same essential approach as encoded bitmap indexes, except that bit-sliced indexes do not need the mapping table. Instead the index encodes the numeric values on its own bit representations.

The Bit-Sliced Index on the C column of table T is the set of all bitmaps Bi, such that bit n of bitmap Bi is set to D (n, i), where, D (n, i) is the value of the ith bit from the right of column C in row n (where the rightmost bit is treated as bit 0 rather than bit 1).

Each individual bitmap Bi is called a *Bit-Slice* of the column. A Bit-Sliced Index (called Bit-Wise Indexing in Sybase IQ [Ren97]) stores a set of "Bitmap slices" which are *orthogonal* to the data held in a Projection index.

Join Indexes [OG95]

A join index is not a fundamental index type. The technique can be used with many types of indexes, including bitmap or B+ tree indexes. [OG95] proposed what it calls *multi-table joins through bitmapped join indices.* A join index is the result of creating an index from more than one table on a join attribute based on predefined joins. Precalculating such indexes avoids computing the actual join at query processing time. Join indexes can be view as a fully pre-computed join or a materialized view. See [OG95] for variations in implementing join indexes. However, a single update in one of the joined tables would require updating all join indexes that involve that table. Join indexes have been implemented in Sybase IQ and Oracle 7.3 [Orl96].

Analysis of Simple Bitmap and Traditional B+ Tree indexes

Traditional B+ tree indexes are the most widely used technique in relational database systems; simple bitmap indexes are a new and popular indexing technique for low cardinality data in data warehouses.

Both indexes have advantages. Traditional B+ trees are dense indexes in the form of disk-resident trees. Their search time complexity is $\log_B(N)$, where B is the number of entries in each node, and N is total number of tuples. Because memory sizes were small, and the price was high, B+ trees were originally developed from a disk-oriented perspective. Since B+ trees can reduce disk I/O during search, they are very suitable for large volumes of high cardinality data. A good B+ tree should look very fat and short. Although building B+ trees is moderately complex, search time is reduced dramatically. These characteristics have made B+ trees popular for regular DBMSs. On the other hand, the simplicity of the bitmap index makes it suitable for the DWH environment. Because of its simple format, it is easy to implement, and no disk I/O is needed if the bitmap can be held in memory. Therefore, for modern computers, high speed bit operations and parallelism can be used to accelerate query processing.

However, both techniques also have drawbacks. Simple bitmap indexing is good for low cardinality data, but not for high cardinality data. The space requirement is N*L bits where L is the number of rows and N is the number of distinct values of the index fields. Normally, as N grows in the fact table, L becomes huge, resulting in a very high value of the product of N and L. While the traditional B tree is good for high cardinality data in conventional databases, it is not suitable for DWH systems. Furthermore, because it is implemented as a dense index, a regular B-tree would not be small enough to fit in memory.

In response to the shortcoming of these two techniques, this paper proposed a modified B+ tree index. The proposed technique not only solves the simple bitmap's space problem for high cardinality data, but also retains the computational efficiency of bitmap indexes. The primary goal is to create a bitmap vector dynamically in the least possible time from a modified B+ tree according to certain query conditions.

A modified B+ tree index should be:

- specially designed for high cardinality data, (low cardinality data would be better implemented in simple bitmap format.);
- easily manipulated in combination with other bitmaps (cooperative), and able to generate bitmaps dynamically during query processing in a short time; and
- small enough to be stored in memory, which suggests that it should be a sparse index, as opposed to the dense index of regular B+ trees.

Chapter 3

Modified B+ Tree Index

Desirable properties of indexing techniques for DWHs

Before evaluating indexing techniques, and before proposing a given technique, the characteristics of DWHs must be evaluated to establish the most important aspects of an indexing system for DWH.

A good indexing technique for DWH should:

- Fit in main memory.

  Accessing and retrieving millions of records from different tables requires large amounts of time due to the amount of disk I/O needed. Search time decreases if the index is kept in memory, or at least in the disk cache, so that disk I/O would not be necessary until the relevant records were finally retrieved.

- Take advantage of the characteristics of the DWHs

  DWH indexes need not work well in regular databases (e.g., bitmap). Since the data in DWHs has characteristics different from that in conventional databases, we can customize the indexing technique for DWHs. In the DWH environment, no insert or update operations are performed on the fact table, although occasionally some of the oldest data will be purged as out-of-date (for instance some companies only keep data within the past 5 years). New tuples will only be appended to the end of the fact table, giving freedom to design an index that only takes append operations into account (no insert, and no update operation.) For example, if insertion were allowed for the fact table, bitmap indexes would be extremely hard to maintain.

- Cooperate readily with other indexing techniques.

  A good indexing technique for DWH should be able to cooperate with other indexing techniques as well. Otherwise, general efficiency will be lost because of the probability of different indexing methods being used on different tables. For example, bitmap indexes are very fast for low cardinality data, while regular B+ tree indexes are suited for high cardinality data. However, there is no way for a regular B+ tree to work with bitmap vectors. A bitmap can be developed under some conditions, but a lot of data must be scanned and retrieved from the disk in order to do this. So cooperation is also critical to establish whether an indexing technique is efficient in broad DWH context.

- Give priority to time complexity over space complexity.

  Although the degree of wasted space in simple bitmap index for high cardinality data cannot be tolerated, in general, the time cost of complex query management with B+ tree indexes is more important. We need to consider time-space trade-offs carefully in evaluating.

A modified B+ tree index on a tale T for field F is constructed in two steps:

1. Create a sorted projection table (F, RID) by projecting out the field F of each row with its RID to a projection table. The projections are then sorted based on the value of F; duplicates should be kept.
2. Construct a modified B+ tree on the projection table, in which the first value of F on each disk physical block is indexed. The leaf node of this B+ tree contains these values with pointers to the associated blocks.

   The following example illustrates the process of creating the modified B+ tree index. Suppose we want to construct a modified B+ tree on field "First Name" for the table in Figure 1.

Row ID	**First Name**	**Gender**	**Major**	**Age**
1	David	M	MIS	23
2	Alan	M	CS	25
3	Carol	M	EE	31
4	Barbara	F	LIS	24
5	David	M	ME	18
6	Amanda	F	IE	30
7	Barbara	F	MIS	19
8	Alan	M	IE	21
*	*	*	*	*

Figure 1

Step1— Create the sorted projection table

The sorted projection table is given in Figure 2:

Name	**Row ID**
Alan	2
Alan	8
*	*
Amanda	6
*	*
Barbara	4
Barbara	7
*	*
Carol	3
*	*
David	1
David	5

Figure 2

The actual structure on the disk of the projection table is give in Figure 3:

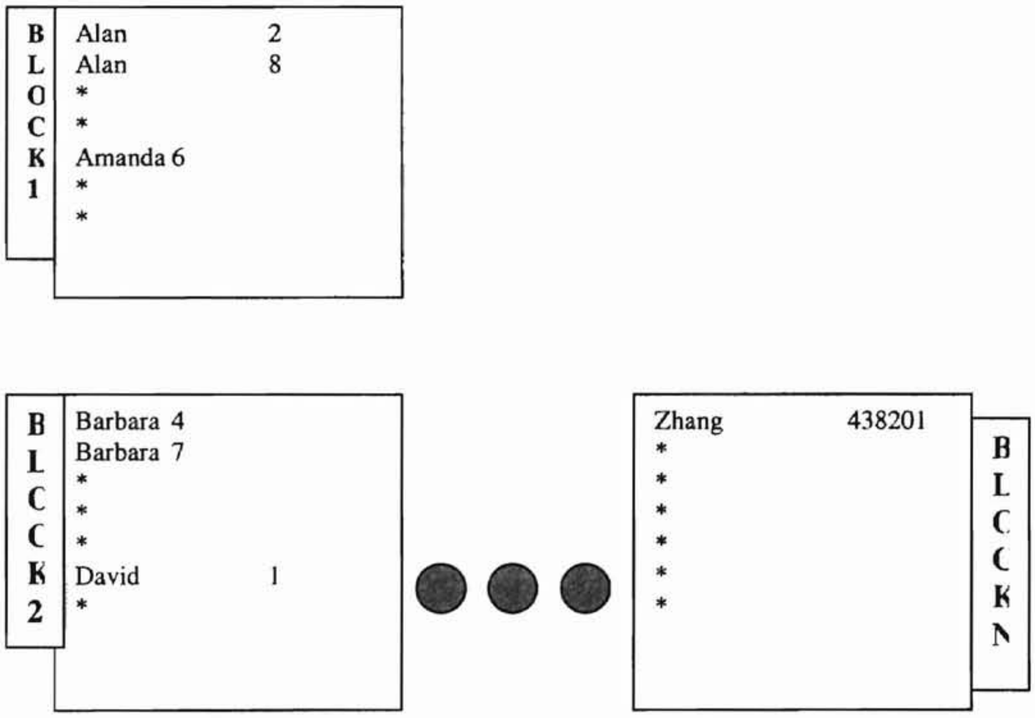

Figure 3

STEP 2 – Create the B+ tree index on the structure of the projection table

The modified B+ tree index is created according to the first value of each block; there should be N leaf nodes where N is the total number of the blocks used in the projection table. In this example, the left-most leaf node should be *Alan* with a pointer to block #1; the second leaf node should be *Barbara* with a pointer to block #2; the right-most leaf node is Zhang, pointing to block #N.

How the modified B+ tree works

The following query illustrates how the modified B+ tree index works.

Query: *Find all the people in Computer Science with a name like JO* whose gender is male.*

Solution:

- Gender and Department have bitmap vectors representing Male and Computer Science that are already loaded memory.
- Generate a bitmap vector for the condition <name like "JO*">. First, generate a blank bitmap vector (with all bits 0) at the length of the number of rows in the table. Search the modified B+ tree index for field 'name', which is already in the memory, and look for: N1-- the leftmost block of entry for JO* (e.g., JN*), and N2—the rightmost block of entry for JO*(e.g., JP*). Then read in all the blocks between N1 and N2, scan all the name entries, and set the corresponding bit to 1 if it is "JO*".
- Finally, perform an "AND" operation on the three bitmap vectors, Name, Gender and Department. The bit positions with 1 in the result vector point to the rows that need to be retrieved as the result.

The projection table is important in modified B+ tree indexes. The projection table is sorted and organized by sequential blocks and associates each value with the RID in the fact table. It inherits the advantages of the projection index, and the RID is used to create the bitmap vector at run time.

Insertion can cause problems, because the projection table RIDs must be re-organized. Fortunately, we don't need to worry about that because of the static attributes of data in DWHs. New data only get appended to the fact table periodically; the newly appended rows can be indexed by our modified B+ tree easily and will not affect the rows that have already been indexed. Only when deletion is performed, does the entire index have to be reconstructed, but this only happens at infrequent intervals (for instance, when the DWH manager wants to purge old records).

Modified B+ Tree Join index for the Star Schema

The Star Schema model consists of a fact table in the center with many dimensional tables associated with it. Join operations happen frequently between the fact table and these dimensions. Our modified B+ tree index has another advantage for constructing the join index between the fact table and dimensional tables.

The projection table used by modified B+ tree provides benefits, not only for indexing the fact table, but also for join indexes for the various dimensional tables. The following example illustrates the concept.

Suppose that a modified B+ tree index has already been created on the Name field of the fact table. This allows a projection table, named P1, to be created on the Name attribute from the fact table. A query condition like "Customer.name = Fact.name and

Customer.age = 18) would be common in a DWH; the join operation on "Name" will happen frequently. A join index would be well suited here. We can construct a join index from the projection table P1. The simplest way is to add a field "Join Block" to the dimension table named Customer table as in Figure 4.

In the Customer table, the "Join Block" field points to the block in which the value of the Name field for that row resides on the projection table P1. For example, "Cindy" first appears in block 8 of the fact table (all the other rows with name "Cindy" in the fact table should be next to each other after the first "Cindy," because the projection table P1 is sorted), so the Join Block field of the row with the name "Cindy" in the Customer points to block #8 of projection table P1.

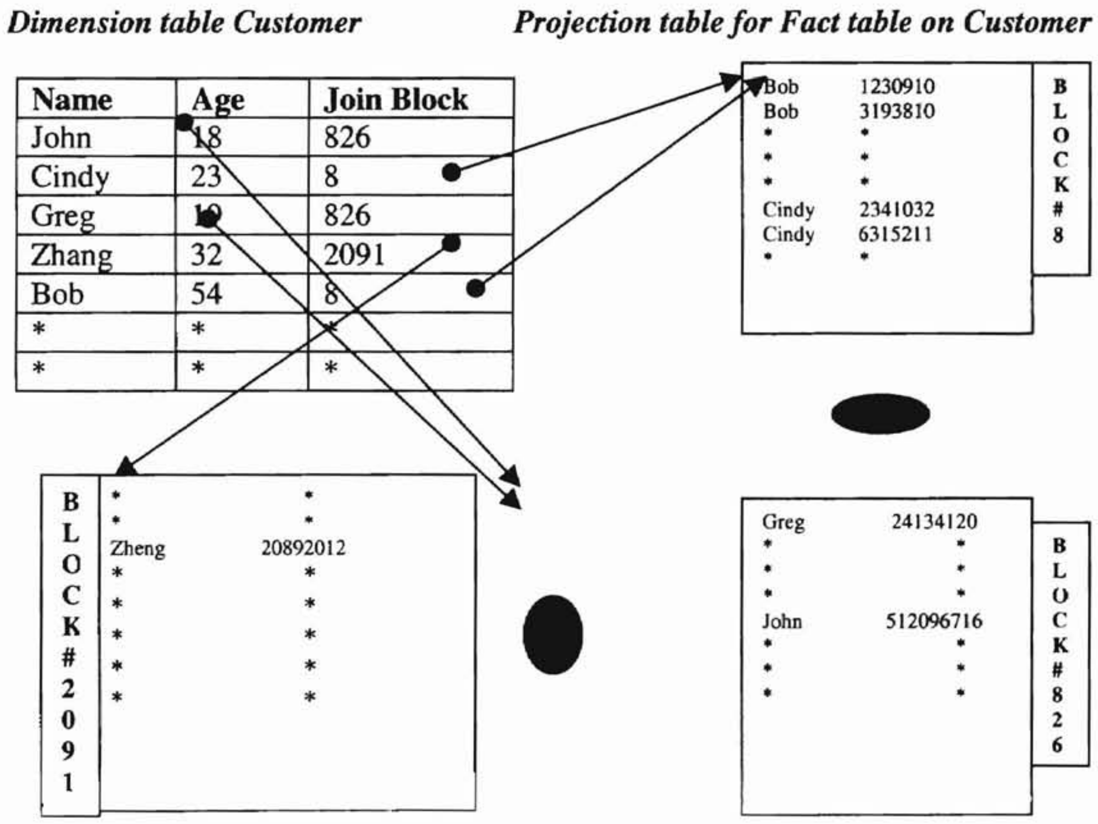

Figure 4

The example above illustrates how the projection table can be used by a dimensional table as the basis for creating a join index. The join index can also be constructed in the format of the modified B+ tree index instead of adding a "Join block" field to each row. The leaf nodes of this tree contain each distinct attribute value and the block number. The difference between this modified B+ tree join index and the modified B+ tree index for the fact table is that the join index actually maps the block according to the different absolute values, while the B+ tree for the fact tables maps only the first value in each block. For example, suppose we have a field entitled "Name" with values "Alan, Amanda, John, Josh… ". A modified B+ join index has leaf nodes with all the values, "Alan", "Amanda", "John", "Josh", etc. But the modified B+ index for the fact table has leaf nodes with only the values of the first entry in each block of the projection table. For example, it may have only "Alan" and "John", without "Amanda" and "Josh"; it may also have "Alan", twice in the index if there are many rows with the name of "Alan" in the fact table. In fact, we use the join index as the index for the fact table, so we don't need to create two indexes (for instance, if each distinct value in the fact table appears many times). As long as the actual space used by the modified B+ tree is not too large, these two trees are for different purposes, and it is not critical to discard one of them.

Chapter 4

Simulation

This study evaluated indexing techniques using Benchmark APB-1 proposed by the OLAP council [Bul96]. The advantage of the APB-1 is that it measures the performance of the database server. It models the DWH using Star Schema, in which multiple table joins are very common among the ten queries. The benchmark simulates a realistic OLAP business situation.

Tables needed

Fact tables: Measure

Dimensional table: Product, Customer, and Time.

Since the Channel and Scenario dimensions have no other attributes and are low cardinality, they are entered into the Measure table as two fields.

The APB-1 benchmark data come in several different forms, depending on the cardinality and size of the data involved. These studies chose the minimum value, which corresponds to a 50 megabyte fact table.

The procedure to evaluate space needs for indexing is as follows. First, the program checks the cardinality of each field for every table. That information is used to derive the space needed to construct a modified B+ tree index and simple bitmap index for these three dimensions.

Fact table: total records No. is 1377000

Product	Customer	Channel

Dimensional Table	Simple Bitmap Index	Regular B+ Tree Index	Modified B+ Tree Index
Product	20 giga bytes		

Dimensional Table	Bitmap Joined Index	Modified B+ Tree Joined Index
Product	20 giga bytes	400 mega bytes
Channel	11 mega bytes	400 mega bytes
Customer	2 giga bytes	400 mega bytes

Figure 5

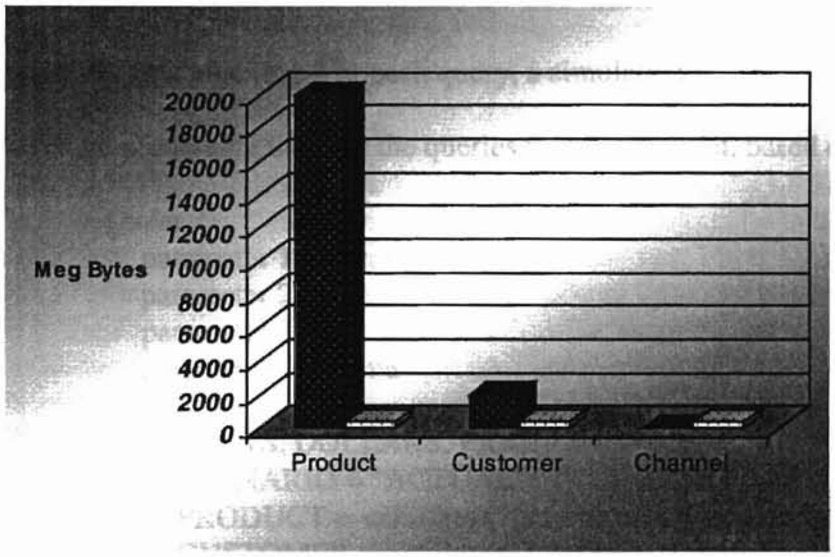

Figure 6

From Figure 5 and Figure 6, we can see that the modified B+ tree index uses about the same amount of disk space for all 3 dimensions. The reason is that most of the space consumed by indexing is for construction of the projection table. The actual B+ tree uses only a small amount of the disk space because it is a sparse index. This small B+ tree resides in memory to reduce the time for searching an entry. Furthermore, using simple bitmap index to create a 20 Gigabyte index for the join index on product is not possible. The cardinality of the Product is 10,000, Customer is 1,000, and Channel is 7 in this simulation.

Because the space complexity of the modified B+ tree index does not depend on the cardinality of the field it indexes, it solves the space waste problem of the simple bitmap indexing technique.

Query performance of the Modified B+ tree and Traditional B+ tree

The query performance of the modified B+ tree index was also compared with conventional B+ tree indexing. For each query, a simulation was performed 1,000 times and the average processing time for the queries was determined, based on [Hua95].

Query 1:

```
parameter 1 = ?product
parameter 2 = ?customer
parameter 3 = ?channel
parameter 4 = ?time

get UNITS, DOLLARS, PRICE
by SCENARIO = "ACTUAL"
by PRODUCT = <children(?product)> option suppress null
by CUSTOMER = <children(?customer)> option suppress null
by CHANNEL = <?channel>
by TIME =<children(?time)>
```

Query 3:

```
parameter 1 = ?product
parameter 2 = ?customer

get UNITS, DOLLARS, COST, INVENTORY
by SCENARIO = "ACTUAL"
by PRODUCT = <?product>
by CUSTOMER = <children(?customer)>
by CHANNEL = attribute ("level", "TOP")
by TIME = "9501" through "9606"
```

Query Processing Plan for Query 1

Proposed Method	Regular B+ Tree method
Assumption: Bitmap vectors for Scenario and Channel already in memory. Modified B+ tree index for fact table and Join index for Product, Time and Customer. 1. Allocate memory for four blank bitmap vectors for Product, Customer and Time dimension, and one for the final result bitmap. Each bitmap is of the length L, where L is the number of rows in the Fact table 2. Go to Dimensional tables Product, Customer and Time, find all the children of the parameters for each of them, according to the Modified B+ tree Join index; set the associated bit positions to 1 in these three bitmap vectors. 3. After dynamically generating these three bitmap vectors for Product, Customer and Time, form the logical AND of those vectors together with the bitmaps for Channel and Time to produce the result vector. 4. According to the bit 1 in the final result bitmap, retrieve and output the result.	Assumption: Regular B+ tree index on (Product, Customer, Time, Channel, Scenario). And Three dimensions have the regular B+ tree index in its dimensional key. 1. Allocate memory for the search key array combinations of (Product, Customer, Time. Channel, Scenario). 2. Go to Dimensional tables Product, Customer and Time, find all the children for each of them, combine them to get the search Key values. 3. Use these search key values to search in the regular B+ tree for the fact table. 4. Retrieve the corresponding rows for output. * Search of the regular B+ tree involves disk I/O.

Query Processing Plan for Query 3

Proposed Method	Regular B+ Tree method
Assumption: Bitmap vectors for Scenario and Channel already in memory. Modified B+ tree index and Join index for Product, Time and Customer. 1. Allocate memory of four blank bitmap vectors for Product, Customer and Time dimension, and one for the final result bitmap. Each bitmap is of the length L. 2. Go to Dimensional tables Product, Customer and Time, find the Product and set the Product bitmap (It only has one Product). Search the modified B+ tree time index for fact table, generate the time bitmap corresponding to "9501" to "9606". Search one-dimensional table Customer for all the children and set up the Customer Vector. 3. AND these five bitmap vectors together to get the final result bitmap. 4. According to the bit 1 in the final result bitmap, retrieve and output the result.	Assumption: Regular B+ tree index on (Product, Customer, Time, Channel, Scenario). And Three dimensions have the regular B+ tree index in its dimensional key. 1. Allocate memory for the search key array Combinations of (Product, Customer, Time. Channel, Scenario). 2. Go to Dimensional table Customer, find all the values of the children, combine them with the product, channel, scenario and time("9501" to "9606") to get the search Key values. 3. Use these search key values to search in the regular B+ tree for the fact table. 4. Retrieve the existed correspondent rows for output. * Search of the regular B+ tree involves disk I/O.

Simulation Result

Figure 7 illustrates the average processing time for one thousand trials of each query.

	Query 1	*Query 3*
Traditional B+ Tree	221,646 ms	9,970 ms
Modified B+ Tree	25,889 ms	5,250 ms

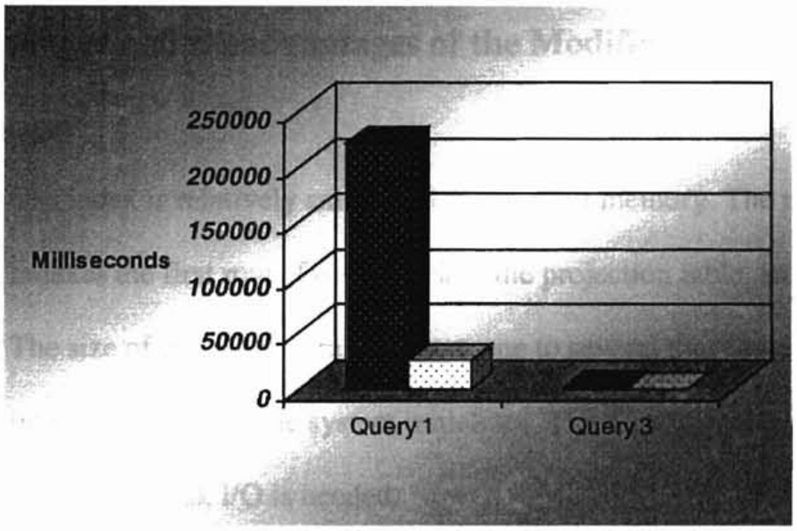

Figure 7

The most important difference between query 1 and query 3 is that in query 1, three dimension tables (Product, Customer and Time) need to be retrieved in order to find all the children of a certain value. But in query 3, time is given, as is product. Only one dimension table, "Customer," must be evaluated. Query 1 has the join operation on all three dimensions tables, but query 3 only joins with "Customer," which is not a very big table.

Also, the foundations of these two queries differ in size. Query 1 has tens of thousands of combinations of values from three dimensions, but Query3 only relates to several hundred. So the disk I/O used by the regular B+ tree searching is limited in query 3.

For the above two reasons, query 1 shows a much greater difference in processing time between the two indexing methods than query 3 does. Query 1 represents queries in DWHs better, since it is complicated, accesses more tables, and combines more values than query 3.

Advantages and Disadvantages of the Modified B+ Tree Index

Advantages:

- The index is relatively small, and fits in main memory. The modified B+ tree only indexes the first row of each block in the projection table, so it is a sparse index. The size of this B+ tree ranges from one to several megabytes, so it can be read into memory when the system initializes. The time to search this tree is very small because no disk I/O is needed.
- The cooperation problem (compared with the regular B+ tree) is solved. Space use is optimized (compared with the simple bitmap index for high cardinality data).
- The number of disk I/O for the range search is reduced, because the sorted projection table supports high-speed range search bitmap vector created at run time.
- It is easy to maintain for appended data, which frequently appears in the fact table.

Disadvantages:

- The projection table needs space. If the projected fields occupy a large portion of the size of each tuple, for example 20%, the projection table would be 20% of the size of the fact table.
- It is not easy to initially construct or reconstruct the projection table and the index since this is very time consuming (i.e., project, sort and generate the B+ tree).

Chapter 5

Conclusion and Future Work

Different indexing techniques are suitable for different situations, depending on the data quantity, data cardinality and common query requirements. However, the situation of the DWH requires that traditional indexing techniques be modified to suit very large data sets and complex query processing. The modified B+ tree indexing technique seems to be suited for the DWH environment because it combines the benefits of both B+ tree and bitmap indexing techniques. Several different indexes can be constructed for the same field to accommodate different query needs. Also the ability to work well with different kinds of index is essential when many different indexing techniques are working together. Finally, the goal of indexing is to speed up query processing, especially the complex iterative queries that commonly occur in the DWH environment. Modified B+ tree indexes have been shown to meet these needs.

However, more research needs to be done. First, B+ trees normally do not reside in main memory. Although we can bring the modified B+ tree into memory, the representation of the tree in main memory should be considered. A sorted array or linked list may be a good memory structure. Last, simulation needs to be done in more detail, not only to compare modified B+ tree indexing with regular B+ tree indexing, but also to compare it with other indexing techniques for high cardinality data (like encoded bitmap indexing).

References

[GRLP96] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh, *Data Cube: A relational aggregation operator generalizing group-by, cross-tabs and sub-totals.*, In Proc. of the 12th Int'l Conference on Data Engineering, pages 152-159, 1996.

[BS96] C.J. Bontempo, C.M. Saracco, *Accelerating Indexed Searching*, Database Programming and Design, pages120-130, July 1996.

[Bul96] D. Bulos. *The APB-1 benchmark*, The OLAP Council benchmark, April 1996, http://www.olapcouncil.org/research/bmarkly.htm.

[CD96] S. Chaudhuri and U. Dayal, *Decision Support, Data Warehousing and OLAP*, 1996 VLDB Tutorial, Bombay, India.

[Ren97] M. Rennhackkamp, *Sybase Warehousing*, DBMS, August 1997

[HRU96] V. Harinarayan, A. Rajaraman, and J.D. Ullman, *Implementing Data Cube Efficiently*, In Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, pages 135-148, Jun.1996.

[WB97] M.C. Wu, A. Buchmann, *Encoded Bitmap Indexing for Data Warehouses*, DVS1, Computer Science Department, Technische University, 1997

[Hua95] J. Huang, *Recovery Techniques in real-time main memory database*, Ph.D Dissertation, Page 150, University of Oklahoma, pages 89,1995.

[AAD+96] S. Agarwal, R. Agarwal, P.M. Deshpande, A. Gupta, J.F. Naughton, R. Ramakrishnan, and S. Sarawagi, *On the computation of multidimensional aggregates.*, In Proc. of the 22nd Int'l Conference on Very Large Databases, pages 506-521, Mumbai (Bombay), India, Sept. 1996.

[OG95] P. O'Neil and G. Graefe, *Multi-Table joins through Bitmapped join indices*, SIGMOD Record, Vol. 24, No. 3, pages 68,Sep. 1995

[OQ97] P. O'Neil, and D. Quass, *Improved Query Performance with Variant Indexes*, SIGMOD Conf., Tucson, Arizona, pages 110,May 1997.

[Orl96] R. J. Orli *Database-Updates to the Bestiary*, 1996, by KISMET Company.

APPENDIX A   Code for checking the cardinality of the data file

```
#include <stdio.h>
#include <stdlib.h>
#define MAXIUM 1024*1024
char unique[MAXIUM][24];
long  num = 0;

void main(int argc, char *argv[])
{
    char f_name[32];
    char str[1024];
    FILE *fp;
    int  begin = 0;
    int  length = 0;
    int  lineno = 0;
    long total_lines = 0;


   printf("\n\n***********************************************************\n");
    printf ("*          Check ..... written by Zhengrong Yi          *\n");
    printf ("*                                                       *\n");
    printf ("* Purpose: To check the cardinality of the datafile     *\n");
    printf ("*                                                       *\n");
    printf ("*                                                       *\n");
    printf ("*                                                       *\n");
    printf ***********************************************************\n\n\n");

    if ( argc != 4 ){
        printf("Syntax Error: check <filename> <begin> <length>\n");
        exit(-1);
    }
    strcpy(f_name,argv[1] );
    begin = atoi( argv[2] );
    begin--;
    length = atoi( argv[3] );
    if (length > 24 ){
        printf("Can't handle string length more than 24!\n");
    }


    fp = fopen(f_name, "r");
    if ( fp == NULL ){
        printf("Can't open the file: %s!!!\n",f_name);
        exit(-2);
    }
    printf("Beginning to Check the cardinality for file:%s,
        begins from %d of length %d\n",f_name,begin,length);

    while ( fgets(str,1024,fp) != NULL ){
        total_lines ++;
        if (mygets(str,begin,length) == 1 ){
            if((num%100) == 0 ){
                printf("Has found %d unique string!\n",num);
            }
            num++;
            if(num > MAXIUM){
                printf("Already exceeds the maxmum number can handle!\n");
                exit(-3);
            }
        }
    }
    printf("Result of Checking the cardinality for file:%s,
        begins from %d of length %d\n",f_name,begin,length);
    printf("Total %d unique strings!\n",num);
    printf("Total %d lines in this file!\n",total_lines);
    fclose( fp );
}

int mygets( char *str,int begin,int length)
{
    char temp[24];
    long i;
    int found = 1;
    for ( i = 0; i < length ;i++){
        temp[i] = str[i+begin];
    }
    temp[length] = 0;
    for ( i = 0; i <num ; i++){
        if ( strcmp(temp,unique[i])==0){
            found = 0;
            break;
        }
    }
    if ( found == 1 ){
        strcpy(unique[num],temp);
    }
    return found;
}
```

APPENDIX B FACT TABLE SIMULATION CODE

```
#include <stdio.h>
#include <stdlib.h>

struct field        // Struct for field;
{
 char f_name[24];      // field name;
 long card;           // the Cardinality
 long max_no;          // Maximum number of tuples in the table that has one certain
value;
 long min_no;          // Minimum number of tuples that has one certain value;
 long average_no;      // if not equals 0, means the number of a certain value is fixed
 int  length;         // the length of the field, in bits
 struct field *next_field; // point to the next field
};

struct table{
     struct field *first; // the first field of the table
     int no;     // total no of the fields in the table
     char table_name[32];
     long tuple_no;
};

void print_table( struct table *table);
int initialize_tables();
void fill_field( struct field *target,char * name, long card, long max,
                    long min, long aver, int length, struct field * next);
```

```
#include <stdio.h>
#include <stdlib.h>
#include "table.h"

extern struct table *Fact;
extern struct table *Customer;
extern struct table *Product;
extern struct table *Time;
extern struct table *Inventory;

int initialize_tables()
{
     struct field *field,*tmp;

     printf("\nInitializing all the tables, please wait.........!\n\n\n");

    /* THE FOLLOWING INITIALIZE THE FACT TABLE FOR SIMULATION*/

     Fact = ( struct table * ) malloc ( sizeof( struct table ));
     if (Fact == NULL ){
          printf("Can not allocate enough memory!\n");
          return -1;

     }
     field = (struct field*) malloc ( sizeof(struct field));
     if (field == NULL ){
printf("Can't allocate enough memory !\n");
          return -1;

     }
     tmp = (struct field*) malloc ( sizeof(struct field));
     if (tmp == NULL ){
printf("Can't allocate enough memory !\n");
          return -1;

     }
     Fact->first = field;
     Fact->no = 6;
     Fact->tuple_no = 1377000;
     strcpy(Fact->table_name, "Fact Table");


     fill_field(field,"CUSTOMER",900,100000,10000,0,12*8,tmp);

     field = tmp;
     tmp = (struct field*) malloc ( sizeof(struct field));
```

```
if (tmp == NULL ){
      printf("Can't allocate enough memory !\n");
      return -1;

}
fill_field(field,"PRODUCT",9000,100000,10000,0,12*8,tmp);

field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory !\n");
      return -1;

}
fill_field(field,"CHANNEL",9,500000,200000,0,12*8,tmp);

field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory !\n");
      return -1;

}
fill_field(field,"TIME",17,500000,200000,0,12*8,tmp);

field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory !\n");
      return -1;

}
fill_field(field,"SCENARIO",3,0,0, 4000000,10*8,tmp);


field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory !\n");
      return -1;

}
fill_field(field,"UNIT_SALES",10000,500000,200000,0,10*8,tmp);


field = tmp;
```

```
    fill_field(field,"DOLLAR_SALES",20000,500000,200000,0,10*8,NULL);


    /* THE FOLLOWING INITIALIZE THE DIMENSION TABLE PRODUCT FOR
SIMULATION */

    Product = ( struct table * ) malloc ( sizeof( struct table ));
    if (Product == NULL ){
        printf("Can't allocate enough memory !\n");
        return -1;

    }

    field = (struct field*) malloc ( sizeof(struct field));
    if (field == NULL ){
        printf("Can't allocate enough memory !\n");
        return -1;

    }
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory !\n");
        return -1;

    }
    Product->first = field;
    Product->no = 3;
    Product->tuple_no = 10000;
    strcpy(Product->table_name, "Product Dimension Table");
    fill_field(field,"MEMBER",10000,0,0,1,12*8,tmp);

    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory !\n");
        return -1;

    }
    fill_field(field,"PARENT",1001,1000,1,0,12*8,tmp);

    field = tmp;
    fill_field(field,"LEVEL",7,9000,1,0,7*8,NULL);
    /* THE FOLLOWING INITIALIZE THE DIMENSION TABLE CUSTOMER FOR
SIMULATION   */

        Customer = ( struct table * ) malloc ( sizeof( struct table ));
        if (Customer == NULL ){
            printf("Can't allocate enough memory !\n");
            return -1;

        }
        field = (struct field*) malloc ( sizeof(struct field));
        if (field == NULL ){
            printf("Can't allocate enough memory !\n");
            return -1;

        }
        tmp = (struct field*) malloc ( sizeof(struct field));
        if (tmp == NULL ){
            printf("Can't allocate enough memory!\n");
            return -1;
}
        Customer->first = field;
        Customer->no = 3;
        Customer->tuple_no = 1000;
        strcpy(Customer->table_name, "Customer Dimension Table");
        fill_field(field,"MEMBER",1000,0,0,1,12*8,tmp);

        field = tmp;
        tmp = (struct field*) malloc ( sizeof(struct field));
        if (tmp == NULL ){
            printf("Can't allocate enough memory!\n");
            return -1;

        }
        fill_field(field,"PARENT",101,100,10,0,12*8,tmp);

        field = tmp;
        fill_field(field,"LEVEL",2,900,1,0,7*8,NULL);


    /* THE FOLLOWING INITIALIZE THE DIMENSION TABLE TIME FOR
SIMULATION   */

        Time=( struct table * ) malloc ( sizeof( struct table ));
        if (Time == NULL ){
            printf("Can't allocate enough memory!\n");
            return -1;
    }
    field = (struct field*) malloc ( sizeof(struct field));
    if (field == NULL ){
          printf("Can't allocate enough memory!\n");
          return -1;

    }
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
          printf("Can't allocate enough memory!\n");
          return -1;

    }
    Time->first = field;
    Time->no = 3;
    Time->tuple_no = 34;
    strcpy(Time->table_name, "Time Dimension Table");
    fill_field(field,"MEMBER",34,0,0,1,12*8,tmp);

    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
          printf("Can't allocate enough memory!\n");
          return -1;

    }
    fill_field(field,"PARENT",9,0,0,4,12*8,tmp);

    field = tmp;
    fill_field(field,"LEVEL",3,1,4,0,12*8,NULL);


  /* THE FOLLOWING INITIALIZE THE DIMENSION TABLE INVENTORY FOR
SIMULATION  */

    Inventory=( struct table * ) malloc ( sizeof( struct table ));
    if (Time == NULL ){
          printf("Can't allocate enough memory!\n");
          return -1;

    }
    field = (struct field*) malloc ( sizeof(struct field));
    if (field == NULL ){
          printf("Can't allocate enough memory!\n");
```

```
        return -1;

    }
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    Inventory->first = field;
    Inventory->no = 19;
    Inventory->tuple_no = 243000;
    strcpy(Inventory->table_name, "Inventory Dimension Table");
    fill_field(field,"CUSTOMER",900,0,0,270,12*8,tmp);

    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"PRODUCT",9000,0,0,27,12*8,tmp);

    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199501",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199502",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;
```

```
}
fill_field(field,"INVENT199503",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
      return -1;

}
fill_field(field,"INVENT199504",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
      return -1;

}
fill_field(field,"INVENT199505",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
      return -1;

}
fill_field(field,"INVENT199506",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
      return -1;

}
fill_field(field,"INVENT199507",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
      return -1;

}
fill_field(field,"INVENT199508",27,0,0,9000,10*8,tmp);
field = tmp;
tmp = (struct field*) malloc ( sizeof(struct field));
if (tmp == NULL ){
      printf("Can't allocate enough memory!\n");
```

```
        return -1;

    }
    fill_field(field,"INVENT199509",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199510",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199511",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199512",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199601",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
    if (tmp == NULL ){
        printf("Can't allocate enough memory!\n");
        return -1;

    }
    fill_field(field,"INVENT199602",27,0,0,9000,10*8,tmp);
    field = tmp;
    tmp = (struct field*) malloc ( sizeof(struct field));
```

```
        if (tmp == NULL ){
              printf("Can't allocate enough memory!\n");
              return -1;

        }
        fill_field(field,"INVENT199603",27,0,0,9000,10*8,tmp);

        field = tmp;
        tmp = (struct field*) malloc ( sizeof(struct field));
        if (tmp == NULL ){
              printf("Can't allocate enough memory!\n");
              return -1;

        }
        fill_field(field,"INVENT199604",27,0,0,9000,10*8,tmp);
        field = tmp;
        fill_field(field,"INVENT199605",27,0,0,9000,10*8,NULL);


      /*********** Testing ***************/

      print_table(Fact);
      print_table(Product);
      print_table(Customer);
      print_table(Time);
      print_table(Inventory);

        return 0;
}

void fill_field( struct field *target,char * name, long card, long max,
                        long min, long aver, int length, struct field * next){
      strcpy(target->f_name,name);
      target->card = card;
      target->max_no= max;
      target->min_no = min;
      target->average_no = aver;
      target->length = length;
      target->next_field = next;


}

void print_table( struct table *table)
{
    int i ;
    struct field *field;
    field = table->first;


    printf("table structure of [%s]\n",table->table_name);
    printf("----------------------------------------------------------------------\n ");
    printf("Name            Cardinality     Length    Max       Min       Average\n");
    printf("----------------------------------------------------------------------\n ");
    for( i = 0; i < table->no ; i++ ){
        printf("%-20s%-20d%-7d%-12d%-11d%-10d\n",field->f_name,field->card,
field->length
                ,field->max_no,field->min_no,field->average_no);
        field = field->next_field;
        if ( field == NULL )
            break;
    }
    printf("----------------------------------------------------------------------\n ");
    printf("Enter to continue!\n");
    getchar();
}
```

APPENDIX C Code for comparing Query performance

```
#include <stdio.h>
#include <stdlib.h>

double retrieve_data (long total_entry);
long get_random(long min, long max);
void query1_execute(int run_times);
double allocate_bitmap( long length);
double and_bitmap( long length,int no);
long get_random_tuple_no(  struct table *table, char *attrib);
double scanbitmap( long length);
double join_child(long fact_ave_no);
double allo_memo_bt( long no, struct table *table, char *attrib);
double set_memo_bt( long no, struct table *table, char *attrib);




#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include "parameter.h"
#include "table.h"
#include "query.h"

extern struct table *Fact;
extern struct table *Customer;
extern struct table *Product;
extern struct table *Time;
extern struct table *Inventory;



long get_random(long min, long max)
{
 long ran;
 double rate;

 ran = rand();
 rate = (double)(ran)/(double)(RAND_MAX);
 ran = min + (max-min)*rate;
 return ran;
}

double allocate_bitmap( long length)  // simulation function for allocating Bitmap
memory
{                         // Input: no. of tuples in a table Output: time needed

 return (  (double)(length/(IO_PAGE_SIZE*8)) * (double)(ALLO_RELA_TIME));
}


double and_bitmap( long length,int no)  // simulation function for And Bitmap vectors
{                         // Input: no. of tuples in a table Output: time needed
                          //      no. of bitmaps needs to AND
double access_time;
double operation_time;

access_time = 2*(((double)(length)/(double)(32))*(double)(MEMO_ACCE_TIME));
operation_time = ((double)(length)/(double)(2))*(1/CPU);
return (no*(access_time+operation_time));
}

void query1_execute(int run_times)
{


/**************************************
Query 1: get UNITS,DOLLARS,PROCE
      by SCENRIO = "ACTULA"
      by PRODUCT = <children(?product)>
      by CUSTOMER = <children(?customer)>
      by CHANNEL = <?channel>
by TIME = <children(?time)>
**************************************/

 double total_btree=0.0 ;
 double total_bitmap=0.0 ;
 double all_btree=0.0 ;
 double all_bitmap=0.0 ;

 long product_no;
 long customer_no;
 long time_no;
long fact_product_no;
long fact_customer_no;
long fact_time_no;
long total_entry;

int i,j;

srand(time(NULL));  // setup random generator
printf("\nBeginning a simulation for query 1...\n");

for ( j = 0; j < run_times ; j++ ){
   printf("\rRunning for No.%i times",j+1);

    total_bitmap = 0.0; // reset the time counter for each round
    total_btree = 0.0;

    total_btree += (double)(INIT_TRAN);  // time to initialize the transaction
    total_bitmap += (double)(INIT_TRAN);



   /* Query processing introduction for our schema
     Suppose that channel bitmap(cardinality of 3 ) has already in the memory,
     the same to the SCENARIO
     Need to create 5 bitmaps for Product, Customer and Time joined from
     three dimension tables Product,Customer and Time, after then, these 5 bitmaps
     will be AND together to get the result
   */

   // setup some random parameter for both methods here
   product_no = get_random_tuple_no( Product,"PARENT");
   customer_no = get_random_tuple_no(  Customer, "PARENT");
   time_no = get_random_tuple_no(  Time, "PARENT");

   fact_product_no = get_random_tuple_no( Fact,"PRODUCT");
   fact_customer_no = get_random_tuple_no(  Fact, "CUSTOMER");
   fact_time_no = get_random_tuple_no(  Fact, "TIME");

   // First allocate 3 bitmaps in memory for Time, Product,
   // and Customer

   total_bitmap += allocate_bitmap( Fact->tuple_no);
   total_bitmap += allocate_bitmap( Fact->tuple_no);
   total_bitmap += allocate_bitmap( Fact->tuple_no);

   // Join from Product Dimension
```

```
total_bitmap += scanbitmap( Product->tuple_no);
for (i = 0; i < product_no ; i++ ){
      total_bitmap += join_child(fact_product_no);
}
// Join from Customer Dimension
total_bitmap += scanbitmap( Customer->tuple_no);
for (i = 0; i < customer_no ; i++ ){
      total_bitmap += join_child(fact_customer_no);
}
// Join from Time Dimension
total_bitmap += scanbitmap( Time->tuple_no);
for (i = 0; i < time_no ; i++ ){
      total_bitmap += join_child(fact_time_no);
}


// At last, And the bitmaps together
total_bitmap += and_bitmap( Fact->tuple_no, 5 );
// Finalize time
total_bitmap += OUTP_TRAN;

/* Query processing introduction for B+ Tree method
First find the product,time and customer childen values
from the dimensional table PRODUCT,TIME and CUSTOMER, then
join these values together with Channel and scenarios as
the key values, search the B+ tree for the fact table,
and get the result
*/

// First allocate memory for Time, Product channel children values
total_btree+= allo_memo_bt( product_no, Product, "MEMBER");
total_btree+= allo_memo_bt( customer_no, Customer, "MEMBER");
total_btree+= allo_memo_bt( time_no, Time, "MEMBER");

// get the children from the Product
total_btree+= SEARCH_BT;
total_btree+= set_memo_bt(product_no, Product, "MEMBER");

// get the children from the Time
total_btree+= SEARCH_BT;
total_btree+= set_memo_bt(time_no, Time, "MEMBER");

// get the children from the Customer
total_btree+= SEARCH_BT;
total_btree+= set_memo_bt(customer_no, Customer, "MEMBER");
```

```
    // combine the values to a key
    // First allocate the memory
    total_btree+= ALLO_RELA_TIME *( (double)(60) *
(double)(product_no*time_no*customer_no) /((double)(IO_PAGE_SIZE)));
    //second, move in these key vales;
    total_btree+= ( (double)(product_no*time_no*customer_no)*36)/((double)(2)) *
(1/CPU);

    // Search these key values in the B Tree
/*
    total_entry = ((product_no*time_no*customer_no)/AVEG_SELE)* IO_PAGE_SIZE
;
    total_btree +=
((double)(total_entry)/AVEG_MOVE)*(SEEK_TIME+LATE_TIME);
    total_btree += (double)(total_entry)/((double)(2))*(1/CPU);
*/
    total_entry = product_no*time_no*customer_no ;
    total_btree += (double)(total_entry)*SEARCH_BT;


    // Finalize time
    total_btree += OUTP_TRAN;

    // make up the time of retrieve the actual data from the disk
    total_btree += retrieve_data (total_entry/AVEG_SELE);
    total_bitmap += retrieve_data (total_entry/AVEG_SELE);
    all_bitmap += total_bitmap;
    all_btree += total_btree;
    printf("** OURS: %f ms!** B+ TREES: %f
ms",(total_bitmap/1000),(total_btree/1000));

  }

  printf("\n\nAverage running time for Query 1 using our method is %f
ms!\n",(all_bitmap/(run_times*1000)));
  printf("Average running time for Query 1 using B+ Tree method is %f
ms!\n",(all_btree/(run_times*1000)));

}
```

```
void query3_execute(int run_times)
{

/*************************************
Query 3: get UNITS,DOLLARS,PRICE
      by SCENRIO = "ACTULA"
      by PRODUCT = <?product>
      by CUSTOMER = <children(?customer)>
      by CHANNEL = attribute("level","TOP")
      by TIME = "9501" thru "9606"
*************************************/


 double total_btree=0.0 ;
 double total_bitmap=0.0 ;
 double all_btree=0.0 ;
 double all_bitmap=0.0 ;

 long customer_no;
 long total_entry;
 long fact_customer_no;

 int i,j;

 srand(time(NULL));  // setup random generator
 printf("\nBeginning a simulation fo query 1...\n");

 for ( j = 0; j < run_times ; j++ ){
    printf("\rRunning for No.%i times",j+1);

     total_bitmap = 0.0; // reset the time counter for each round
     total_btree = 0.0;

     total_btree += (double)(INIT_TRAN); // time to initialize the transaction
     total_bitmap += (double)(INIT_TRAN);


    /* Query processing introduction for our schema
      Suppose that channel bitmap(cardinality of 3 ) has already in the memory
      the same to the SCENARIO
      Need to create 3 bitmaps for Product, Customer and Time ,after then, these 5
bitmaps
      will be AND together to get the result, the product and the customer bitmap will
AND
      to get the combination to retrieve the data from the Inventory table
    */
```

```
// setup some random parameter for both methods here
customer_no = get_random_tuple_no( Customer, "PARENT");
fact_customer_no = get_random_tuple_no( Fact, "CUSTOMER");

// First allocate 4 bitmaps in memory for Time, Product,
// Customer and the result of AND operation on Customer and Product

total_bitmap += allocate_bitmap( Fact->tuple_no);
total_bitmap += allocate_bitmap( Fact->tuple_no);
total_bitmap += allocate_bitmap( Fact->tuple_no);

// Join from Product Dimension
total_bitmap += SEARCH_BT;

// Join from Customer Dimension
total_bitmap += join_child(1);

for (i = 0; i < customer_no ; i++ ){
      total_bitmap += join_child(fact_customer_no);
}
// Join from Time Dimension

total_bitmap += 17* scanbitmap( Time->tuple_no);



// And the bitmaps Product and Customertogether
total_bitmap += and_bitmap( Fact->tuple_no, 2 );

// And with the rest bitmaps together
total_bitmap += and_bitmap( Fact->tuple_no, 2 );

total_entry = 17*customer_no ;

// create the dynamic bitmap on Inventory
total_bitmap += total_entry * SEARCH_BT;



// Finalize time

total_bitmap += retireve_data (total_entry/AVEG_SELE);
total_bitmap += retireve_data (total_entry/AVEG_SELE);
total_bitmap += OUTP_TRAN;
```

```
    /* Query processing introduction for B+ Tree method
    First find the customer childen values
    from the dimensional table CUSTOMER, then
    join these values together with Channel and scenarios as
    the key values, search the B+ tree for the fact table,
    and get the result
    */

    // First allocate memory for Time, Product channel children values
    total_btree+= allo_memo_bt( customer_no, Customer, "MEMBER");

    // get the children from the Customer
    total_btree+= SEARCH_BT;
    total_btree+= set_memo_bt(customer_no, Customer, "MEMBER");

    // combine the values to a key
    // First allocate the memory
    total_btree+= ALLO_RELA_TIME *( (double)(60) * (double)(17*customer_no)
/((double)(IO_PAGE_SIZE)));
    //second, move in these key vales;
    total_btree+= ( (double)(17*customer_no)*36)/((double)(2)) * (1/CPU);

    // Search these key values in the B Tree
    total_btree += (double)(total_entry)*SEARCH_BT;

    // Finalize time
    total_btree += retrieve_data (total_entry/AVEG_SELE);
    total_btree += retrieve_data (total_entry/AVEG_SELE);
    total_btree += OUTP_TRAN;

    // make up the time of retrieve the actual data from the disk
    total_btree += retrieve_data (total_entry/AVEG_SELE);
    all_bitmap += total_bitmap;
    all_btree += total_btree;
    printf("** OURS: %f ms!** B+ TREES: %f
ms",(total_bitmap/1000),(total_btree/1000));

  }

  printf("\n\nAverage running time for Query 1 using our method is %f
ms!\n",(all_bitmap/(run_times*1000)));
  printf("Average running time for Query 1 using B+ Tree method is %f
ms!\n",(all_btree/(run_times*1000)));

}
```

```
double set_memo_bt( long no, struct table *table, char *attrib)
{
	int i;
	struct field *field;
	int found = 0;
	double time=0;

	field = table->first;
	for( i = 0; i < table->no ; i++ ){
		if ( strcmp(field->f_name,attrib) == 0 ){
			found = 1;
			break;
		}
		field = field->next_field;
		if ( field == NULL )
			break;
	}

	if ( found == 1 ){
		time+= (field->length*no/32) *(1/CPU);
	}
	else {
		beep(3);
		printf("Error: Can't find the attribute of %s of table %s\n",
			attrib,table->table_name);
		exit(-1);
	}
	return time;


}

double allo_memo_bt( long no, struct table *table, char *attrib)
{
	int i;
	struct field *field;
	int found = 0;
	double time=0;

	field = table->first;
	for( i = 0; i < table->no ; i++ ){
		if ( strcmp(field->f_name,attrib) == 0 ){
			found = 1;
			break;
		}
		field = field->next_field;
```

```
          if ( field == NULL )
               break;
     }

     if ( found == 1 ){
          time+= ALLO_RELA_TIME *( (double)(field->length/8) * no
/((double)(IO_PAGE_SIZE)));
     }
     else {
          beep(3);
          printf("Error: Can't find the attribute of %s of table %s\n",
               attrib,table->table_name);
          exit(-1);
     }
     return time;


}

long get_random_tuple_no(  struct table *table, char *attrib)
{
     int i;
     long no=0;
     struct field *field;
     int found = 0;

     field = table->first;
     for( i = 0; i < table->no ; i++ ){
          if ( strcmp(field->f_name,attrib) == 0 ){
               found = 1;
               break;
          }
          field = field->next_field;
          if ( field == NULL )
               break;
     }

     if ( found == 1 ){
          if ( field->average_no == 0 )
               no = get_random(field->min_no, field->max_no);
          else no = field->average_no;
     }
     else {
          beep(3);
          printf("Error: Can't find the attribute of %s of table %s\n",
               attrib,table->table_name);
```

```
            exit(-1);
        }
        return no;

}

double join_child(long fact_ave_no)
{
        double t=0;

        t+=SEARCH_BT;
        t+=(double)(fact_ave_no/2)*(double)(1/CPU);
        return t;
}
double scanbitmap( long length)
{
        return ( ((double)(length)/16.0)*
((double)(MEMO_ACCE_TIME)+(double)(1/(2*CPU))));
}


double retrieve_data (long total_entry)
{
        double tt ;

        tt += SEEK_TIME;
        tt += LATE_TIME;
        tt += total_entry * TRAN_PAGE_TIME ;
        return tt;

}
```

APPENDIX D Simulation Program for Indexing Techniques in Data Warehouse

```
#include <stdio.h>
#include <stdlib.h>

#define CPU     140.0      // CPU Power of 140M HZ
#define SEEK_TIME  10000.0  // Average Seek time
#define ALLO_RELA_TIME 5.0  // Allocate/Release a main memory page time
#define INIT_TRAN 100.0    // Average Initialization time of a transaction
#define OUTP_TRAN 100.0    // Average Output time of a transaction
#define IO_PAGE_SIZE 23476  // Page size bytes
#define MEMO_ACCE_TIME 0.18 // main memory access time per word
#define TRAN_PAGE_TIME 64.0  // time to transfer 1 data page
#define LATE_TIME    5560.0 // Latency time
#define AVEG_MOVE    100000.0 // assume 100k bytes needs 1 MOVE of head
#define ROW_ID      32  // Row ID size, 4 Bytes
#define SEARCH_BT    10.0  // Average search time for a memory resident B tree
#define AVEG_SELE    50  // 1/50 of selection results 1 page I/O




/******************************************************************
* Simulation Program for  Indexing Techniques in  DataWarehouse *
*                                   *
* Instructor: Dr. Terry Nutter                 *
* Student:Zhengrong Yi                         *
*                                   *
* Fall 1999                           *
*******************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "table.h"
#include "parameter.h"
#include "query.h"


void   welcome_msg(); // Function that prints out the welcome message
void   exit_msg();  // Function that prints out the exit message
int    get_selection();
int    get_runtimes();
int    beep(int times);
int    mygetint();
int    mygetchar();
void   underconstruction();



int run_times=0;
struct table *Fact;
struct table *Customer;
struct table *Product;
struct table *Time;
struct table *Inventory;


void main()
{
      int selection;
      int out = 0; // Tag for exit the program


      welcome_msg();
      initialize_tables();

      while ( 1 ){

            selection = get_selection();
            if (( selection == 'l') || ( selection == 'L')){
                  out = 1;
            }
            if ( out == 1 )
                  break;
            run_times = get_runtimes();
            switch(selection){
                  case 'a':
                  case 'A':
                        printf ("\n\nYou have select Automatic Execution of queries for %d
times\n",run_times);
                        underconstruction();
                        break;
                  case 'b':
                  case 'B':
                        printf ("\n\nYou have select Execution of queries 1 for %d
times\n",run_times);
                        query1_execute(run_times);
                        break;
                  case 'c':
```

```
            case 'C':
                  printf ("\n\nYou have select Execution of queries 2 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'd':
            case 'D':
                  printf ("\n\nYou have select Execution of queries 3 for %d
times\n",run_times);
                  query3_execute(run_times);
                  break;
            case 'e':
            case 'E':
                  printf ("\n\nYou have select Execution of queries 4 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'f':
            case 'F':
                  printf ("\n\nYou have select Execution of queries 5 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'g':
            case 'G':
                  printf ("\n\nYou have select Execution of queries 6 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'h':
            case 'H':
                  printf ("\n\nYou have select Execution of queries 7 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'i':
            case 'I':
                  printf ("\n\nYou have select Execution of queries 8 for %d
times\n",run_times);
                  underconstruction();
                  break;
            case 'j':
            case 'J':
                  printf ("\n\nYou have select Execution of queries 9 for %d
times\n",run_times);
                  underconstruction();
```

```
                    break;
                case 'k':
                case 'K':
                    printf ("\n\nYou have select Execution of queries 10 for %d
times\n",run_times);
                    underconstruction();
                    break;
                case 'l':
                case 'L':
                    out = 1;
                    break;
                default:
                    beep(3);
                    printf("Wrong selection\n");
                    break;
            }
        }
    exit_msg();
}

void   welcome_msg()
{

printf("\n****************************************************************
*\n");
    printf("* Simulation Program for  Indexing Techniques in DataWarehouse  *\n");
    printf("*                                                      *\n");
    printf("* Instructor: Dr.Terry Nutter                                   *\n");
    printf("* Student: Zhengrong Yi                                         *\n");
    printf("*                                                      *\n");
    printf("* Fall 1999                                              *\n");

printf("*****************************************************************\
n");
    printf("Press Enter to continue!\n");
    getchar();
    printf("\n\n\n\n\n\n\n\n\n");
}





void   exit_msg()
{
```

```
printf("\n*****************************************************************
*\n");
   printf("*                                                  *\n");
   printf("*          (\"'-"-/\").___..--"\"'-._                 *\n");
   printf("*           '6_6 )  '-. (    ).'-.__.')              *\n");
   printf("*            (_Y_.)' ._  ) '._ '. ''-..-'            *\n");
   printf("*          _..'--'_..-_/ /--'_.','                  *\n");
   printf("*        (il),-" (li),' ((!,-'                      *\n");

printf("******************************************************************\
n");
   printf("* Goodbye!!!! Thanks for testing this program              *\n");

printf("******************************************************************\
n");
   printf("Press Enter to continue!\n");
   getchar();
   printf("\n\n\n\n\n\n\n\n\n");
}


int   get_selection()
{
   char user_selection;

   while ( 1 ){
    printf("+------------------------------+\n");
    printf("|      QUERY SELECTION         |\n");
    printf("+------------------------------+\n");
    printf("|  A.---Automatic execution    |\n");
    printf("|  B.---Execute Query 1        |\n");
    printf("|  C.---Execute Query 2        |\n");
    printf("|  D.---Execute Query 3        |\n");
    printf("|  E.---Execute Query 4        |\n");
    printf("|  F.---Execute Query 5        |\n");
    printf("|  G.---Execute Query 6        |\n");
    printf("|  H.---Execute Query 7        |\n");
    printf("|  I.---Execute Query 8        |\n");
    printf("|  J.---Execute Query 9        |\n");
    printf("|  K.---Execute Query 10       |\n");
    printf("|  L.---Exit the Program       |\n");
    printf("+------------------------------+\n");
    printf("Please enter your selection:  ");
    user_selection=mygetchar();
    if ( (user_selection <= 'l' ) && ( user_selection >='A' ))
```

```
            break;
        else {
            beep(2);
            printf("%c is an Invalid Selection!\n",user_selection);
        }
      }
      return user_selection;
}

int    beep(int times)
{
        int i,j;
        for ( i = 0 ; i < times ; i++ ){
            putchar(0x7);
            for ( j = 0; j < 100000 ; j++ );
        }
}

int    get_runtimes()
{
        int user_times;
        printf("How many times you want to run the query(queries)? :");
        user_times = mygetint();
        return user_times;

}

int mygetint()
{
        int it = 0;
        int first = 0;
        char str[80];
        while(1){
            gets(str);
            it = atoi(str);
            if ( it <=0  ){
                if ( first  != 0 ){
                    beep(4);
                    printf("\nError input, please reenter a integer number!\n");
                }
                else first = 1;
            }
            else break;
        }
        return it;
}
```

```
int mygetchar()
{

    int i;
    char str[80];
    char ch;
    int found = 0;
    while(1){
        gets(str);
        for (i = 0 ; i < strlen(str) ; i++ ){
            ch = (char)( str[i]);
            if ( (ch <= 'z' ) && ( ch >='A' )){
                found = 1;
                break;
            }
        }
        if ( found == 0 ){
            beep(4);
            printf("\nError input, please reenter a character!\n");
        }
        else break;
    }
    return ch;
}

void  underconstruction()
{
    printf("**********************************************************\n");
    printf("*                  SORRY                               *\n");
    printf("*The Function you selected is not available at this time!*\n");
    printf("*          Press Enter  to continue!                   *\n");
    printf("**********************************************************\n");
    getchar();
}
```

APPENDIX E Simulation program Execution result

Script started on Fri Dec 17 12:31:09 1999

```
/c/yzhengr/sim>> gcc simulation.c
simulation.c: In function 'main':
simulation.c:921: warning: return type of 'main' is not
'int'
/c/yzhengr/sim>> a.out

***********************************************************
******
* Simulation Program for  Indexing Techniques in
DataWarehouse  *
*
*
* Instructor: Dr. Nutter
*
* Student: Zhengrong Yi
*
*
*
* Fall 1999
*
***********************************************************
******
Press Enter to continue!











Initializing all the tables, please wait.........!
```

```
table structure of [Fact Table]
-----------------------------------------------------------
--------------
Name                Cardinality       Length     Max
Min       Average
-----------------------------------------------------------
--------------
CUSTOMER            900                 96     100000
10000       0
PRODUCT             9000                96     100000
10000       0
CHANNEL             9                   96     500000
200000      0
TIME                17                  96     500000
200000      0
SCENARIO            3                   80     0
0           4000000
UNIT_SALES          10000               80     500000
200000      0
-----------------------------------------------------------
--------------
Enter to continue!
```

```
table structure of [Product Dimension Table]
-----------------------------------------------------------
--------------
Name                Cardinality       Length     Max
Min       Average
-----------------------------------------------------------
--------------
MEMBER              10000               96     0
0           1
PARENT              1001                96     1000
1           0
LEVEL               7                   56     9000
1           0
-----------------------------------------------------------
--------------
Enter to continue!
```

```
table structure of [Customer Dimension Table]
-----------------------------------------------------------
-------------
Name                Cardinality       Length     Max
Min      Average
-----------------------------------------------------------
-------------
MEMBER              1000                96     0
0          1
PARENT              101                 96     100
10         0
LEVEL               2                   56     900
1          0
-----------------------------------------------------------
-------------
Enter to continue!

table structure of [Time Dimension Table]
-----------------------------------------------------------
-------------
Name                Cardinality       Length     Max
Min      Average
-----------------------------------------------------------
-------------
MEMBER              34                  96     0
0          1
PARENT              9                   96     0
0          4
LEVEL               3                   96     1
4          0
-----------------------------------------------------------
-------------
Enter to continue!
```

```
table structure of [Inventory Dimension Table]
-----------------------------------------------------------
--------------
Name                Cardinality       Length      Max
Min      Average
-----------------------------------------------------------
--------------
CUSTOMER            900                   96      0
0          270
PRODUCT             9000                  96      0
0          27
INVENT199501        27                    80      0
0          9000
INVENT199502        27                    80      0
0          9000
INVENT199503        27                    80      0
0          9000
INVENT199504        27                    80      0
0          9000
INVENT199505        27                    80      0
0          9000
INVENT199506        27                    80      0
0          9000
INVENT199507        27                    80      0
0          9000
INVENT199508        27                    80      0
0          9000
INVENT199509        27                    80      0
0          9000
INVENT199510        27                    80      0
0          9000
INVENT199511        27                    80      0
0          9000
INVENT199512        27                    80      0
0          9000
INVENT199601        27                    80      0
0          9000
INVENT199602        27                    80      0
0          9000
INVENT199603        27                    80      0
0          9000
INVENT199604        27                    80      0
0          9000
INVENT199605        27                    80      0
0          9000
-----------------------------------------------------------
--------------
```

```
Enter to continue!

+---------------------------------+
|        QUERY SELECTION          |
+---------------------------------+
|    A.---Automatic execution     |
|    B.---Execute Query 1         |
|    C.---Execute Query 2         |
|    D.---Execute Query 3         |
|    E.---Execute Query 4         |
|    F.---Execute Query 5         |
|    G.---Execute Query 6         |
|    H.---Execute Query 7         |
|    I.---Execute Query 8         |
|    J.---Execute Query 9         |
|    K.---Execute Query 10        |
|    L.---Exit the Program        |
+---------------------------------+
Please enter your selection:
......

Error input, please reenter a character!

......

Error input, please reenter a character!
b
How many times you want to run the query(queriess)?  :6


You have select Execution of queries 1 for 6 times

Beginning a simulation fo query 1...

Running for No.1 times** OURS: 592.286553 ms!** B+ TREES:
1256.964061 ms
Running for No.2 times** OURS: 900.431524 ms!** B+ TREES:
2533.071922 ms
Running for No.3 times** OURS: 814.192995 ms!** B+ TREES:
2705.538750 ms
Running for No.4 times** OURS: 307.566567 ms!** B+ TREES:
274.801763 ms
Running for No.5 times** OURS: 318.355324 ms!** B+ TREES:
542.095812 ms
Running for No.6 times** OURS: 306.629024 ms!** B+ TREES:
623.415893 ms

Average running time for Query 1 using our method is
539.910331 ms!
```

```
Average running time for Query 1 using B+ Tree method is
1322.648033 ms!
```

```
+--------------------------------+
|         QUERY SELECTION        |
+--------------------------------+
|   A.---Automatic execution     |
|   B.---Execute Query 1         |
|   C.---Execute Query 2         |
|   D.---Execute Query 3         |
|   E.---Execute Query 4         |
|   F.---Execute Query 5         |
|   G.---Execute Query 6         |
|   H.---Execute Query 7         |
|   I.---Execute Query 8         |
|   J.---Execute Query 9         |
|   K.---Execute Query 10        |
|   L.---Exit the Program        |
+--------------------------------+
Please enter your selection:  d
How many times you want to run the query(queries)?  :3


You have select Execution of queries 3 for 3 times

Beginning a simulation for query 1...

Running for No.1 times** OURS: 188.008489 ms!** B+ TREES:
108.719347 ms
Running for No.2 times** OURS: 177.402560 ms!** B+ TREES:
110.521909 ms
Running for No.3 times** OURS: 168.625996 ms!** B+ TREES:
101.681527 ms

Average running time for Query 1 using our method is
178.012348 ms!
```

```
Average running time for Query 1 using B+ Tree method is
106.974261 ms!
+---------------------------------+
|         QUERY SELECTION         |
+---------------------------------+
|   A.---Automatic execution      |
|   B.---Execute Query 1          |
|   C.---Execute Query 2          |
|   D.---Execute Query 3          |
|   E.---Execute Query 4          |
|   F.---Execute Query 5          |
|   G.---Execute Query 6          |
|   H.---Execute Query 7          |
|   I.---Execute Query 8          |
|   J.---Execute Query 9          |
|   K.---Execute Query 10         |
|   L.---Exit the Program         |
+---------------------------------+
Please enter your selection:  1
```

```
************************************************************
******
*
*
*             ("`-''-/").____..--''"`-._
*
*              `6_ 6  )   `-.  (     ).`-.__.`)
*
*               (_Y_.)'  ._   )  `._ `. ``-..-'
*
*            _..`--'_..-_/  /--'_.' ,'
*
*          (il),-''  (li),'  ((!,-'
*
************************************************************
******
* Goodbye!!!! Thanks for testing this program
*
************************************************************
******
Press Enter to continue!

/c/yzhengr/sim>> exit
script done on Fri Dec 17 12:32:15 1999
```

VITA

Zhengrong Yi

Candidate for the Degree of

Master of Science

Thesis: MODIFIED B+ TREE TECHNIQUE IN DATA WAREHOUSE ENVIRONMENT

Major Field: Computer Science

Biographical:

Personal Data: Born in Hubei, China, on February 22, 1969

Education: Graduated from Jinzhou High School, Jinzhou, city, Hubei, and received the Bachelor's degree in Automation Engineering from Harbin industrial University China in July, 1991. Completed the requirements of the Master of Science at Oklahoma State University in December 2000.

Professional Experience: Employed by Earth Products Instruments (HK) Ltd. Shenzhen, China, as an Engincer, 1992 to 1997; employed by Hipro Electronics, Inc. Austin, Texas, as a Software Co-Engineer, February 2000 to May 2000.