BITMAP INDEXING v/s B^+TREE INDEXING IN DATA WAREHOUSING

A *Major Project Report*

Submitted in partial fulfillment of the requirements for the award of the degree of

Bachelor of Technology

in

COMPUTER SCIENCE and ENGINEERING

By

ASHOK KUMAR YENDA	**12011A0506**
HEMANT KOTI	**12011A0515**
SRIKAR YAGAVANDLA	**12011A0541**

Under the Esteemed Guidance of

Dr. K. P. SUPREETHI
Assistant Professor
Department Of Computer Science And Engineering

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
JNTUH COLLEGE OF ENGINEERING HYDERABAD
(Autonomous)
Kukatpally, Hyderabad – 500 085
May 2016.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JNTUH COLLEGE OF ENGINEERING HYDERABAD

(Autonomous)

KUKATPALLY, HYDERABAD-500 085.

DECLARATION BY THE CANDIDATES

We hereby declare that work described in this Major Project report entitled **"Bitmap Indexing vs B$^+$Tree Indexing in Data Warehousing"** which is being submitted by us in the Department of Computer Science and Engineering, **JNTUH College of Engineering** is the result of studies carried out by us under the guidance of **Dr. K. P. Supreethi**, Assistant Professor of Computer Science and Engineering, JNTUH - CEH, Hyderabad.

This work is original and has not been submitted for any degree or diploma of this or any other university.

ASHOK KUMAR YENDA	**HEMANT KOTI**	**SRIKAR YAGAVANDLA**
12011A0506	**12011A0515**	**12011A0541**

Department of Computer Science and Engineering,

JNTUH College of Engineering

Kukatpally,

Hyderabad - 500085.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JNTUH COLLEGE OF ENGINEERING HYDERABAD

(Autonomous)

KUKATPALLY, HYDERABAD-500 085.

CERTIFICATE BY THE SUPERVISOR

This is to certify that the Major Project report entitled "**Bitmap Indexing vs B^+Tree Indexing in Data Warehousing**", being submitted by **Ashok Kumar Yenda, Hemant Koti**, **Srikar Yagavandla** in partial fulfillment of the requirements for the award of the degree of Bachelor of Technology in **Computer Science and Engineering Department** of Jawaharlal Nehru Technological University Hyderabad College of Engineering, is a record of bonafide work carried out by them during the academic year 2015 – 2016.

The results of the investigation enclosed in this report have been verified and found satisfactory.

Dr. K.P.Supreethi,

Assistant Professor,

Dept. of Computer Science & Engineering,

JNTUH College of Engineering(Autonomous),

Hyderabad - 500085.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JNTUH COLLEGE OF ENGINEERING HYDERABAD

(Autonomous)

KUKATPALLY, HYDERABAD-500 085.

CERTIFICATE BY THE HEAD OF THE DEPARTMENT

This is to certify that the Major Project report entitled "**Bitmap Indexing vs B$^+$Tree Indexing in Data Warehousing**" that is being submitted by **Ashok Kumar Yenda**, **Hemant Koti**, **Srikar Yagavandla** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering** is a record of bonafide work carried out by them during the academic year 2015 – 2016.

Dr. V. Kamakshi Prasad,

Professor & Head of Department,

Dept. of Computer Science & Engineering,

JNTUH College of Engineering(Autonomous),

Kukatpally, Hyderabad - 500085.

ACKNOWLEDGEMENTS

We wish to extend our sincere gratitude to our supervisor **Dr. K. P. Supreethi**, Assistant Professor of the Department**,** for being a driving force all through the way. This project would not be so smooth and so interesting without her encouragement.

We also wish to express our gratitude to **Dr. V. Kamakshi Prasad,** Professor, and Head**,** Department of CSE for providing necessary computing facilities. We are indebted to the Department of Computer Science & Engineering, JNTUH for providing us with all the required facilities to carry our work in a congenial environment.

We also extend our gratitude to the Computer Science & Engineering Department staff for providing necessary support from time to time whenever requested.

Ashok Kumar Yenda (12011A0506)
Hemant Koti (12011A0515)
Srikar Yagavandla (12011A0541)

ABSTRACT

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed. Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

As far as Database Systems was concerned, the previously proposed indexing methods were sufficient but with the invention of Data Warehousing and Data Mining, there were challenges to come up with new techniques that were far more logical and efficient. They are B^{+}Tree Indexing, Bitmap Indexing, Join Index, and Projection Index.

A bitmap index is a preferred indexing technique for cases where the indexed attributes are of few distinct values (i.e., low cardinality). Once the index size is huge, the cardinality of indexed columns increases causing the query response time to rise.

On the other hand, owing to its indexing and retrieving mechanisms, the B^{+}Tree index is assumed to be the adequate technique as the column values increase in cardinality.

This report seeks to illustrate the retrieval time comparison for B^{+}Tree, Bitmap, and sequential search techniques.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

A Data Warehouse (DW) is the foundation for Decision Support Systems (DSS) with a large collection of information that can be accessed through an On-line Analytical Processing (OLAP) application. This large database stores current and historical data that comes from several external data sources. The queries built on DW systems are complex and usually include some join operations that incur computational overhead which rises the response time especially when queries are performed on a large dataset. To increase the performance, DW analysts commonly use some solutions such as indexes, summary tables, and partition mechanism.

There are various index techniques supported by database vendors such as Bitmap, B^+Tree, Projection, Join bitmap, and Range base bitmap indices among others. A Bitmap index for example is advisable for a system comprising data that are not frequently updated by many concurrent processes. This is mainly because a Bitmap index stores large amounts of row information in each block of the index structure. Also, since Bitmap index locking is at the block level, any insert, update or delete activity may result in locking an entire range of values. By contrast, a B^+Tree index is adequate for a system that is frequently updated because it does not need re-balancing as frequently as other self-balancing search trees. Besides, all leaf blocks of the tree are at the same depth. Thus, choosing the proper type of index structures has a significant impact on the DW environment.

The main problem is that there is no definite guideline for DW analysts to choose appropriate indexing methods. According to common practice, the Bitmap index is best suited for columns having low cardinality and should be only considered for low-cardinality data. If the number of distinct values of a column is less than 1%, then the column is a candidate for a Bitmap index. This assumption may be correct to some extent based on previous algorithms and based on old machine processing used by the database software and hardware respectively, but, as the usage of data is exploding, this assumption may no longer be applicable.

1.1 Objective

The main objective of this report is to compare the records retrieval time for both bitmap indexing and B^+Tree indexing. To show this difference, we implement the bitmap indexing algorithm and B^+Tree indexing algorithm in JAVA. Because this is an indexing technique suited for Data Warehouses, we take large data sets to perform the test. Before getting into the actual implementation, we first introduced to the readers the organization of data in the system.

1.2 Scope and Limitations

Through this thesis work, we are trying to sort out the various drawbacks that we face in case of *sequential* retrieval of data in the case of **Data Warehouses**. To enable such efficient retrieval of data, we use the concepts of indexing. **Indexes** are used to quickly locate data without having to search every row in a **database** table every time a **database** table is accessed. **Indexes** can be created using one or more columns of a **database** table, providing the basis for both rapid random lookups and efficient access of ordered records.

The indexing techniques that we have taken up for our study are **BITMAP and B$^+$Tree Indexing.** This paper covers the definition of both the indexing techniques, their creation, and comparison in terms of the time required for the extraction of data from huge databases. Slight improvement to the Bitmap indexing is also proposed (Bitmap Join Indexing) which improves the efficiency further. This indexing technique best works when the database has **low-selectivity.**

After introducing the readers to the basics of both the indexing techniques, we have also illustrated the difference in time between the two techniques. Later the shortcomings of both the indexing techniques are also listed so that the readers can come up with their indexing technique which eliminates such restrictions.

1.3 Organization

In **chapter 1**, we introduce to the readers the basic concepts that are involved in Bitmap Indexing and B$^+$Tree indexing to get them started, the scope of the project i.e its length and breadth are also covered.

In **chapter 2**, the concepts of disk organizations, file organization, page formats, record formats, and the various indexing techniques that are used by the database vendors are explained in depth.

In **chapter 3**, the problem is stated i.e the purpose of using Bitmap indexing and B$^+$Tree indexing over sequential search and retrieval is described. This section also discusses the advantages of Bitmap indexing over the B$^+$Tree indexing technique.

In **chapter 4**, Bitmap indexing and B$^+$Tree indexing are explained in depth by using relational tables and other illustrations to present the idea to the reader. In addition to these techniques, the Bitmap Join Index is also explained.

Chapter 5 includes the tools that were used to create the front end, the back end, and the interaction between the front end and the back end. It also shows the relational tables that were used for testing.

In **chapter 6**, we have plotted the time taken for retrieval with indexing and without indexing to perform the search operations using a bar graph and scatter plot. The output is also displayed through illustrations where sample data was used for testing.

In **chapter 7**, the limitation of this report and the directions for conducting future work are also listed.

In **chapter 8**, the various sources like the IEEE journals, the textbooks that we used for reference are listed.

CHAPTER 2: LITERATURE SURVEY

The system consists of primary memory and secondary memory. The primary memory (main memory) is where our applications and processes run. The secondary memory contains the data for reading and writing information.

Since the secondary memory is very large compared to the primary memory, the operating system uses the concept of "pages" and "file organization". Data from the secondary storage are stored in pages in small chunks which are processed sequentially. File organization helps to store the data in a logical format to reduce the access time. In the world of database systems, we extend this concept by using indexes for efficient retrieval of large data sets. Indexing techniques maintain records that are organized in sequence based on a key field. They are mainly used for random access to data. To further improve the performance, we employ disk organization techniques that essentially contain multiple disks that operate independently and in parallel.

2.1 File Organisation

File organization refers to the logical structuring of the records as determined by how they are accessed.

In choosing a file organization, several criteria are important:

• Short access time

• Ease of update

• Economy of storage

• Simple maintenance

• Reliability

The relative priority of these criteria will depend on the applications that will use the file. For example, if a file is only to be processed in batch mode, with all of the records accessed every time, then rapid access for retrieval of a single record is of minimal concern. A file stored on CD-ROM will never be updated, and so ease of update is not an issue.

These criteria may conflict. For example, for the economy of storage, there should be minimum redundancy in the data. On the other hand, redundancy is a primary means of increasing the speed of access to data. An example of this is the use of indexes. The number of alternative file organizations that have been implemented or just proposed is unmanageably large, even for a book devoted to file systems. In this brief survey, we will outline five fundamental organizations. Most structures used in actual systems either fall into one of these categories or can be implemented with a combination of these organizations.

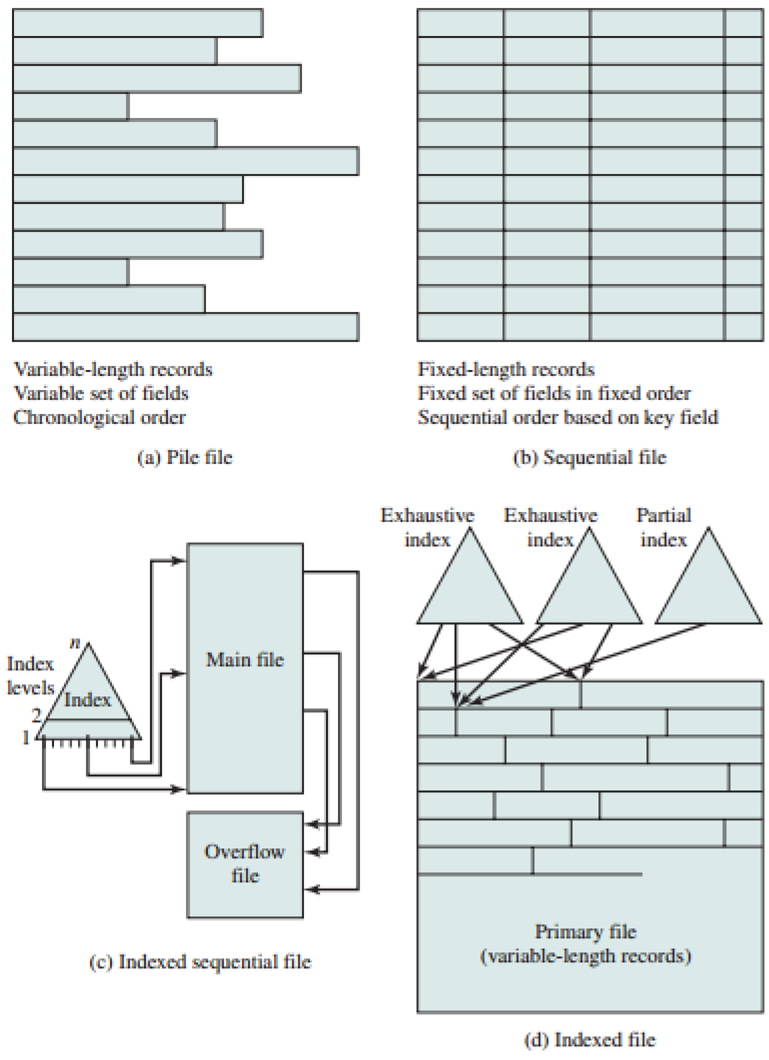
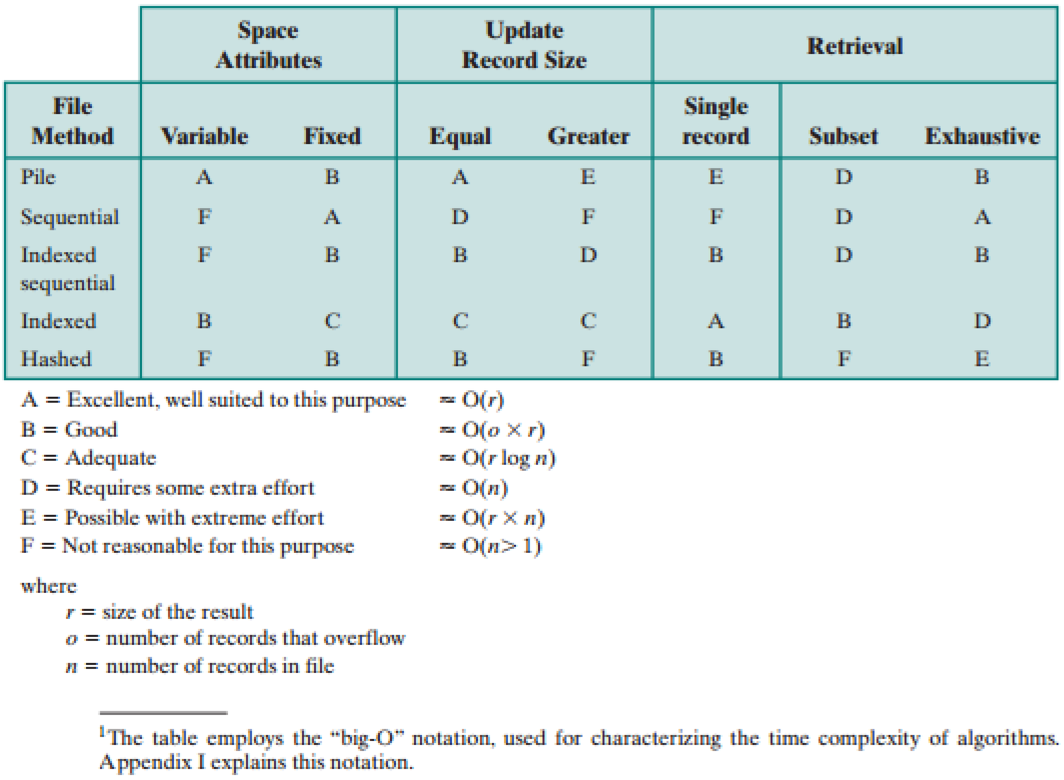


Figure 2.1 Common file Organisation

2.2 The Pile

The least-complicated form of file organization may be termed the pile. Data are collected in the order in which they arrive. Each record consists of one burst of data. The purpose of the pile is simply to accumulate the mass of data and save it. Records may have different fields or similar fields in different orders. Thus, each field should be self-describing, including a field name as well as a value. The length of each field must be implicitly indicated by delimiters, explicitly included as a subfield, or known as default for that field type. Because there is no structure to the pile file, record access is by exhaustive search. That is, if we wish to find a record that contains a particular field with a particular value, it is necessary to examine each record in the pile until the desired record is found or the entire file has been searched. If we wish to find all records that contain a particular field or contain that field with a particular value, then the entire file must be searched. Pile files are encountered when data are collected and stored before processing or when data are not easy to organize. This type of file uses space well when the stored data vary in size and structure, is perfectly adequate for exhaustive searches, and is easy to update. However, beyond these limited uses, this type of file is unsuitable for most applications.

Table 2.1 Grades of Performance of 5 basic File organizations



	Space Attributes		**Update Record Size**		**Retrieval**		
File Method	**Variable**	**Fixed**	**Equal**	**Greater**	**Single record**	**Subset**	**Exhaustive**
Pile	A	B	A	E	E	D	B
Sequential	F	A	D	F	F	D	A
Indexed sequential	F	B	B	D	B	D	B
Indexed	B	C	C	C	A	B	D
Hashed	F	B	B	F	B	F	E

A = Excellent, well suited to this purpose	$\approx O(r)$
B = Good	$\approx O(o \times r)$
C = Adequate	$\approx O(r \log n)$
D = Requires some extra effort	$\approx O(n)$
E = Possible with extreme effort	$\approx O(r \times n)$
F = Not reasonable for this purpose	$\approx O(n > 1)$

where
r = size of the result
o = number of records that overflow
n = number of records in file

[1]The table employs the "big-O" notation, used for characterizing the time complexity of algorithms. Appendix I explains this notation.

2.3 The Sequential File

The most common form of the file structure is the sequential file. In this type of file, a fixed format is used for records. All records are of the same length, consisting of the same number of fixed-length fields in a particular order. Because the length and position of each field are known, only the values of fields need to be stored; the field name and length for each field are attributes of the file structure.

One particular field, usually the first field in each record, is referred to as the key field. The key field uniquely identifies the record; thus key values for different records are always different. Further, the records are stored in key sequence: alphabetical order for a text key, and numerical order for a numerical key.

Sequential files are typically used in batch applications and are generally optimum for such applications if they involve the processing of all the records (e.g., a billing or payroll application). The sequential file organization is the only one that is easily stored on tape as well as disk.

For interactive applications that involve queries and/or updates of individual records, the sequential file provides poor performance. Access requires the sequential search of the file for a key match. If the entire file, or a large portion of the file, can be brought into the main memory at one time, more efficient search techniques are possible. Nevertheless, considerable processing and delay are encountered to access a record in a large sequential file. Additions to the file also present problems. Typically, a sequential file is stored in the simple sequential ordering of the records within blocks. That is, the physical organization of the file on tape or disk directly matches the logical organization of the file. In this case, the usual procedure is to place new records in a separate pile file, called a log file or transaction file. Periodically, a batch update is performed that merges the log file with the master file to produce a new file in a correct key sequence.

An alternative is to organize the sequential file physically as a linked list. One or more records are stored in each physical block. Each block on disk contains a pointer to the next block. The insertion of new records involves pointer manipulation but does not require that the new records occupy a particular physical block position. Thus, some added convenience is obtained at the cost of additional processing and overhead.

2.4 The Indexed Sequential File

A popular approach to overcoming the disadvantages of the sequential file is the indexed sequential file. The indexed sequential file maintains the key characteristic of the sequential file: Records are organized in sequence based on a key field. Two features are added: an index to the file to support random access, and an overflow file. The index provides a lookup capability to reach quickly the vicinity of a desired record. The overflow file is similar to the log file used with a sequential file but is integrated so that a record in the overflow file is located by following a pointer from its predecessor record.

In the simplest indexed sequential structure, a single level of indexing is used. The index in this case is a simple sequential file. Each record in the index file consists of two fields: a key fields which are the same as the key field in the main file, and a pointer into the main file. To find a specific field, the index is searched to find the highest key value that is equal to or precedes the desired key value. The search continues in the main file at the location indicated by the pointer.

To see the effectiveness of this approach, consider a sequential file with 1 million records. To search for a particular key value will require on average one half million record accesses. Now suppose that an index containing 1,000 entries is constructed, with the keys in the index more or less evenly distributed over the main file. Now it will take on average 500 accesses to the index file followed by 500 accesses to the main file to find the record. The average search length is reduced from 500,000 to 1,000.

Additions to the file are handled in the following manner: Each record in the main file contains an additional field not visible to the application, which is a pointer to the overflow file. When a new record is to be inserted into the file, it is added to the overflow file. The record in the main file that immediately precedes the new record in the logical sequence is updated to contain a pointer to the new record in the overflow file. If the immediately preceding record is itself in the overflow file, then the pointer in that record is updated. As with the sequential file, the indexed sequential file is occasionally merged with the overflow file in batch mode.

The indexed sequential file greatly reduces the time required to access a single record, without sacrificing the sequential nature of the file. To process the entire file sequentially, the records of the main file are processed in sequence until a pointer to the overflow file is found, then accessing continues in the overflow file until a null pointer is encountered, at which time accessing of the main file is resumed where it left off.

To provide even greater efficiency in access, multiple levels of indexing can be used. Thus the lowest level of the index file is treated as a sequential file and a higher level index file is created for that file. Consider again a file with 1 million records. A lower-level index with 10,000 entries is constructed. A higher-level index into the lower-level index of 100 entries can then be constructed. The search begins at the higher-level index (average length = 50 accesses) to find an entry point into the lower-level index. This index is then searched (average length = 50) to find an entry point into the main file, which is then searched (average length = 50). Thus the average length of search has been reduced from 500,000 to 1,000 to 150.

2.5 The Indexed File

The indexed sequential file retains one limitation of the sequential file: Effective processing is limited to that which is based on a single field of the file. For example, when it is necessary to search for a record based on some other attribute than the key field, both forms of the sequential file are inadequate. In some applications, the flexibility of efficiently searching by various attributes is desirable.

To achieve this flexibility, a structure is needed that employs multiple indexes, one for each type of field that may be the subject of a search. In the general indexed file, the concept of sequentially and a single key is abandoned. Records are accessed only through their indexes. The result is that there is now no restriction on the placement of records as long as a pointer in at least one index refers to that record. Furthermore, variable-length records can be employed.

Two types of indexes are used. An exhaustive index contains one entry for every record in the main file. The index itself is organized as a sequential file for ease of searching. A partial index contains entries to records where the field of interest exists. With variable-length records, some records will not contain all fields. When a new record is added to the main file, all of the index files must be updated.

Indexed files are used mostly in applications where timeliness of information is critical and where data are rarely processed exhaustively. Examples are airline reservation systems and inventory control systems.

2.6 The Direct or Hashed File

The direct, or hashed, file exploits the capability found on disks to access directly any block of a known address. As with sequential and indexed sequential files, a key field is required in each record. However, there is no concept of sequential ordering here. The direct file makes use of hashing on the key value.

Direct files are often used where very rapid access is required, where fixed-length records are used, and where records are always accessed one at a time. Examples are directories, pricing tables, schedules, and name lists.

2.7 Disk Organisation

The rate of improvement in secondary storage performance has been considerably less than the rate for processors and main memory. This mismatch has made the disk storage system perhaps the main focus of concern in improving overall computer system performance.

As in other areas of computer performance, disk storage designers recognize that if one component can only be pushed so far, additional gains in performance are to be had by using multiple parallel components. In the case of disk storage, this leads to the development of arrays of disks that operate independently and in parallel. With multiple disks, separate I/O requests can be handled in parallel, as long as the data required reside on separate disks. Further, a single I/O request can be executed in parallel if the block of data to be accessed is distributed across multiple disks.

With the use of multiple disks, there is a wide variety of ways in which the data can be organized and in which redundancy can be added to improve reliability. This could make it difficult to develop database schemes that are usable on several platforms and operating systems. Fortunately, the industry has agreed on a standardized scheme for multiple-disk database design, known as RAID (redundant array of independent disks). The RAID scheme consists of seven levels, 2 zero through six. These levels do not imply a hierarchical relationship but designate different design architectures that share three common characteristics:

1. RAID is a set of physical disk drives viewed by the OS as a single logical drive.

2. Data are distributed across the physical drives of an array in a scheme known as striping, described subsequently.

3. Redundant disk capacity is used to store parity information, which guarantees data recoverability in case of a disk failure.

The details of the second and third characteristics differ for the different RAID levels. RAID 0 and RAID 1 do not support the third characteristic.

The term RAID was originally coined in a paper by a group of researchers at the University of California at Berkeley. The paper outlined various RAID configurations and applications and introduced the definitions of the RAID levels that are still used. The RAID strategy employs multiple disk drives and distributes data in such a way as to enable simultaneous access to data from multiple drives, thereby improving I/O performance and allowing easier incremental increases in capacity.

The unique contribution of the RAID proposal is to address effectively the need for redundancy. Although allowing multiple heads and actuators to operate simultaneously achieves higher I/O and transfer rates, the use of multiple devices increases the probability of failure. To compensate for this decreased reliability, RAID makes use of stored parity information that enables the recovery of data lost due to a disk failure.

We now examine each of the RAID levels. Table 2.1 provides a rough guide to the seven levels. In the table, I/O performance is shown both in terms of data transfer capacity, or ability to move data, and I/O request rate, or ability to satisfy I/O requests, since these RAID levels inherently perform differently relative to these two metrics. Each RAID level's strong point is highlighted in color. Figure 2.1 is an example that illustrates the use of the seven RAID schemes to support a data capacity requiring four disks with no redundancy. The figure highlights the layout of user data and redundant data and indicates the relative storage requirements of the various levels. We refer to this figure throughout the following discussion.

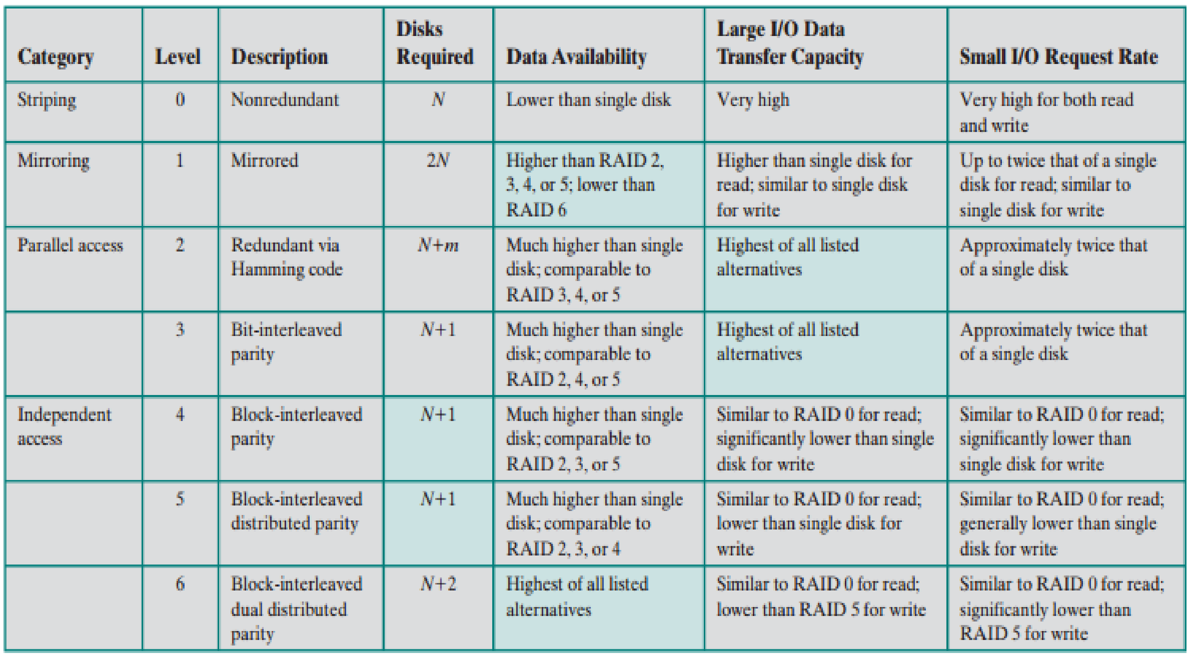
Of the seven RAID levels described, only four are commonly used: RAID levels 0, 1, 5, and 6.

2.7.1 RAID Level 0

RAID level 0 is not a true member of the RAID family, because it does not include redundancy to improve performance or provide data protection. However, there are a few applications, such as some on supercomputers in which performance and capacity are primary concerns and low cost is more important than improved reliability.

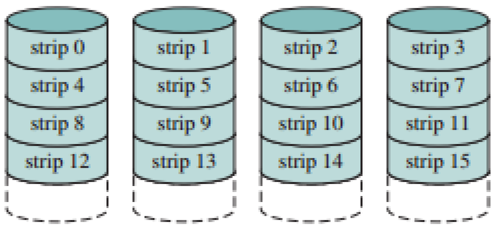
For RAID 0, the user and system data are distributed across all of the disks in the array. This has a notable advantage over the use of a single large disk: If two different I/O requests are pending for two different blocks of data, then there is a good chance that the requested blocks are on different disks. Thus, the two requests can be issued in parallel, reducing the I/O queueing time. But RAID 0, as with all of the RAID levels, goes further than simply distributing the data across a disk array: The data are striped across the available disks.

Table 1.2 Raid Levels

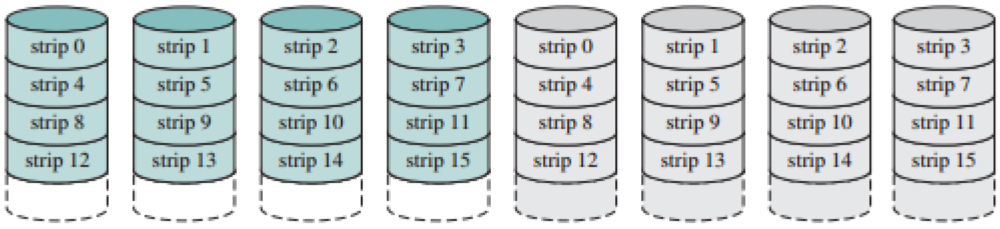


Category	Level	Description	Disks Required	Data Availability	Large I/O Data Transfer Capacity	Small I/O Request Rate
Striping	0	Nonredundant	N	Lower than single disk	Very high	Very high for both read and write
Mirroring	1	Mirrored	$2N$	Higher than RAID 2, 3, 4, or 5; lower than RAID 6	Higher than single disk for read; similar to single disk for write	Up to twice that of a single disk for read; similar to single disk for write
Parallel access	2	Redundant via Hamming code	$N+m$	Much higher than single disk; comparable to RAID 3, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
	3	Bit-interleaved parity	$N+1$	Much higher than single disk; comparable to RAID 2, 4, or 5	Highest of all listed alternatives	Approximately twice that of a single disk
Independent access	4	Block-interleaved parity	$N+1$	Much higher than single disk; comparable to RAID 2, 3, or 5	Similar to RAID 0 for read; significantly lower than single disk for write	Similar to RAID 0 for read; significantly lower than single disk for write
	5	Block-interleaved distributed parity	$N+1$	Much higher than single disk; comparable to RAID 2, 3, or 4	Similar to RAID 0 for read; lower than single disk for write	Similar to RAID 0 for read; generally lower than single disk for write
	6	Block-interleaved dual distributed parity	$N+2$	Highest of all listed alternatives	Similar to RAID 0 for read; lower than RAID 5 for write	Similar to RAID 0 for read; significantly lower than RAID 5 for write

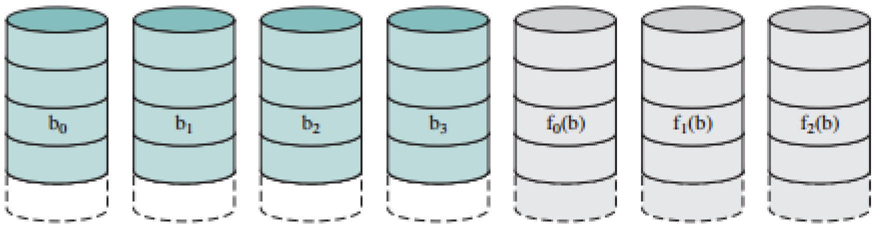
Note: N, number of data disks; m, proportional to log N.



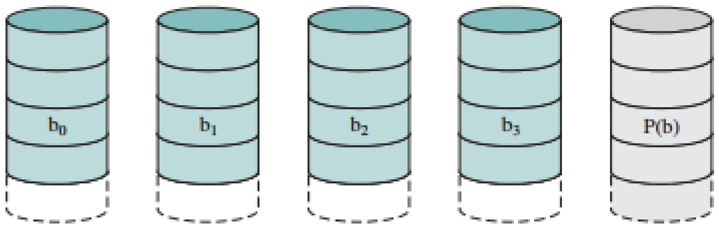
(a) RAID 0 (nonredundant)



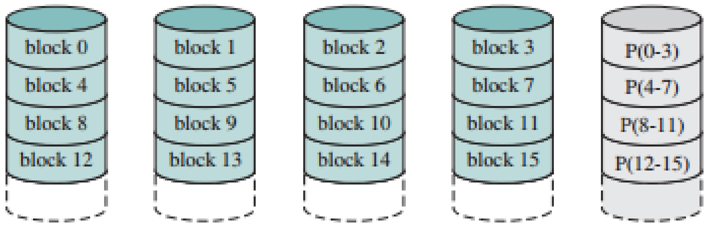
(b) RAID 1 (mirrored)



(c) RAID 2 (redundancy through Hamming code)



(d) RAID 3 (bit-interleaved parity)



(e) RAID 4 (block-level parity)

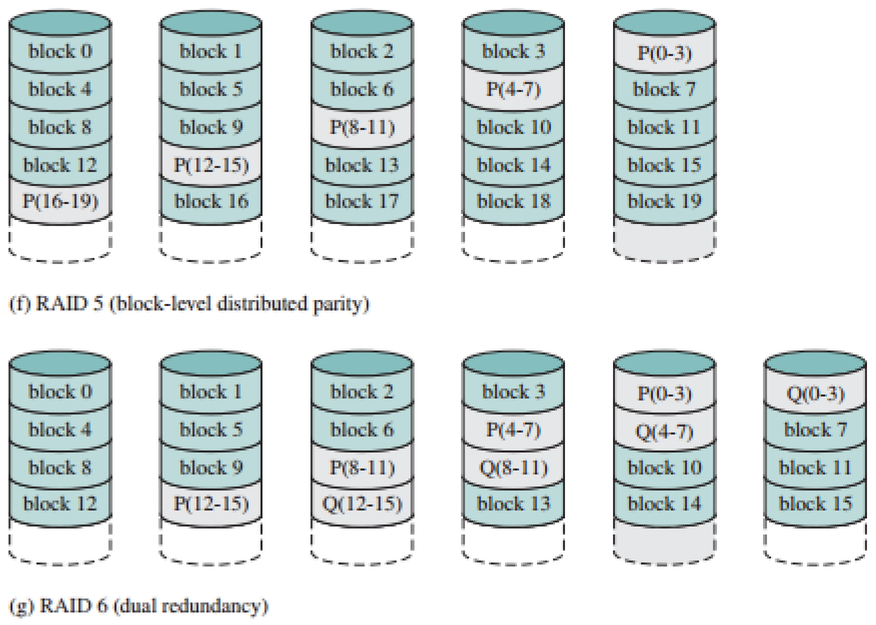


Figure 2.2 Raid Levels

This is best understood by considering Figure 2.2. All user and system data are viewed as being stored on a logical disk. The logical disk is divided into strips; these strips may be physical blocks, sectors, or some other unit. The strips are mapped round-robin to consecutive physical disks in the RAID array. A set of logically consecutive strips that maps exactly one strip to each array member is referred to as a stripe. In an n -disk array, the first n logical strips are physically stored as the first strip on each of the n disks, forming the first stripe; the second n strips are distributed as the second strips on each disk; and so on. The advantage of this layout is that if a single I/O request consists of multiple logically contiguous strips, then up to n strips for that request can be handled in parallel, greatly reducing the I/O transfer time.

Raid 0 For High Data Transfer Capacity

The performance of any of the RAID levels depends critically on the request patterns of the host system and the layout of the data. These issues can be most clearly addressed in RAID 0, where the impact of redundancy does not interfere with the analysis. First, let us consider the use of RAID 0 to achieve a high data transfer rate. For applications to experience a high transfer rate, two requirements must be met. First, a high transfer capacity must exist along the entire path between host memory and the individual disk drives. This includes internal controller buses, host system I/O buses, I/O adapters, and host memory buses.

The second requirement is that the application must make I/O requests that drive the disk array efficiently. This requirement is met if the typical request is for large amounts of logically contiguous data, compared to the size of a strip. In this case, a single I/O request involves the parallel transfer of data from multiple disks, increasing the effective transfer rate compared to a single-disk transfer.

Raid 0 For High I/O Request Rate

In a transaction-oriented environment, the user is typically more concerned with response time than with the transfer rate. For an individual, I/O request for a small amount of data, the I/O time is dominated by the motion of the disk heads (seek time) and the movement of the disk (rotational latency).

In a transaction environment, there may be hundreds of I/O requests per second. A disk array can provide high I/O execution rates by balancing the I/O load across multiple disks. Effective load balancing is achieved only if there are typically multiple I/O requests outstanding. This, in turn, implies that there are multiple independent applications or a single transaction-oriented application that is capable of multiple asynchronous I/O requests. The performance will also be influenced by the strip size. If the strip size is relatively large, so that a single I/O request only involves single disk access, then multiple waiting I/O requests can be handled in parallel, reducing the queuing time for each request.

2.7.2 RAID Level 1

RAID 1 differs from RAID levels 2 through 6 in the way in which redundancy is achieved. In these other RAID schemes, some form of parity calculation is used to introduce redundancy, whereas, in RAID 1, redundancy is achieved by the simple expedient of duplicating all the data. Figure 2.2 b shows data striping being used, as in RAID 0. But in this case, each logical strip is mapped to two separate physical disks so that every disk in the array has a mirror disk that contains the same data. RAID 1 can also be implemented without data striping, though this is less common.

There are several positive aspects to the RAID 1 organization:

1. A read request can be serviced by either of the two disks that contains the requested data, whichever one involves the minimum seek time plus rotational latency.

2. A written request requires that both corresponding strips be updated, but this can be done in parallel. Thus, the write performance is dictated by the slower of the two writes (i.e., the one that involves the larger seek time plus rotational latency). However, there is no “write penalty” with RAID 1. RAID levels 2 through 6 involve the

use of parity bits. Therefore, when a single strip is updated, the array management software must first compute and update the parity bits as well as updating the actual strip in question.

3. Recovery from a failure is simple. When a drive fails, the data may still be accessed from the second drive.

The principal disadvantage of RAID 1 is the cost; it requires twice the disk space of the logical disk that it supports. Because of that, a RAID 1 configuration is likely to be limited to drives that store system software and data and other highly critical files. In these cases, RAID 1 provides real-time backup of all data so that in the event of a disk failure, all of the critical data is still immediately available.

In a transaction-oriented environment, RAID 1 can achieve high I/O request rates if the bulk of the requests are reads. In this situation, the performance of RAID 1 can approach double that of RAID 0. However, if a substantial fraction of the I/O requests are written requests, then there may be no significant performance gain over RAID 0. RAID 1 may also provide improved performance over RAID 0 for data transfer-intensive applications with a high percentage of reads. Improvement occurs if the application can split each read request so that both disk members participate.

2.7.3 RAID Level 2

RAID levels 2 and 3 make use of a parallel access technique. In a parallel access array, all member disks participate in the execution of every I/O request. Typically, the spindles of the individual drives are synchronized so that each disk head is in the same position on each disk at any given time.

As in the other RAID schemes, data striping is used. In the case of RAID 2 and 3, the strips are very small, often as small as a single byte or word. With RAID 2, an error-correcting code is calculated across corresponding bits on each data disk, and the bits of the code are stored in the corresponding bit positions on multiple parity disks. Typically, a Hamming code is used, which can correct single-bit errors and detect double-bit errors.

Although RAID 2 requires fewer disks than RAID 1, it is still rather costly. The number of redundant disks is proportional to the log of the number of data disks. On a single read, all disks are simultaneously accessed. The requested data and the associated error-correcting code are delivered to the array controller. If there is a single-bit error, the controller can recognize and correct the error instantly, so that the read access time is not slowed. On a single write, all data disks and parity disks must be accessed for the write operation.

RAID 2 would only be an effective choice in an environment in which many disk errors occur. Given the high reliability of individual disks and disk drives, RAID 2 is overkill and is not implemented.

2.7.4 RAID Level 3

RAID 3 is organized similarly to RAID 2. The difference is that RAID 3 requires only a single redundant disk, no matter how large the disk array. RAID 3 employs parallel access, with data distributed in small strips. Instead of an error-correcting code, a simple parity bit is computed for the set of individual bits in the same position on all of the data disks.

REDUNDANCY In the event of a drive failure, the parity drive is accessed and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive and operation resumed. Data reconstruction is simple. Consider an array of five drives in which X0 through X3 contains data and X4 is the parity disk. The parity for the i^{th} bit is calculated as follows:

X4(i) = X3(i) X2(i) X1(i) X0(i)

Where X is an exclusive-OR function.

Suppose that drive X1 has failed. If we add X4(i) X1(i) to both sides of the preceding equation, we get

X1(i) = X4(i) X3(i) X2(i) X0(i)

Thus, the contents of each strip of data on X1 can be regenerated from the contents of the corresponding strips on the remaining disks in the array. This principle is true for RAID levels 3 through 6.

In the event of a disk failure, all of the data are still available in what is referred to as reduced mode. In this mode, for reads, the missing data are regenerated on the fly using the exclusive-OR calculation. When data are written to a reduced RAID 3 array, consistency of the parity must be maintained for later regeneration. Return to full operation requires that the failed disk be replaced and the entire contents of the failed disk are regenerated on the new disk.

Performance: Because data are striped in very small strips, RAID 3 can achieve very high data transfer rates. Any I/O request will involve the parallel transfer of data from all of the data disks. For large transfers, performance improvement is especially noticeable. On the other hand, only one I/O request can be executed at a time. Thus, in a transaction-oriented environment, performance suffers.

2.7.5 RAID Level 4

RAID levels 4 through 6 make use of an independent access technique. In an independent access array, each member disk operates independently, so that separate I/O requests can be satisfied in parallel. Because of this, independent access arrays are more suitable for applications that require high I/O request rates and are relatively less suited for applications that require high data transfer rates.

As in the other RAID schemes, data striping is used. In the case of RAID 4 through 6, the strips are relatively large. With RAID 4, a bit-by-bit parity strip is calculated across corresponding strips on each data disk, and the parity bits are stored in the corresponding strip on the parity disk.

RAID 4 involves a write penalty when an I/O write request of a small size is performed. Each time that a write occurs, the array management software must update not only the user data but also the corresponding parity bits. Consider an array of five drives in which X0 through X3 contains data and X4 is the parity disk. Suppose that a write is performed that only involves a strip on disk X1. Initially, for each bit i, we have the following relationship:

$$X4(i) = X3(i)\ X2(i)\ X1(i)\ X0(i)$$

After the update, with potentially altered bits indicated by a prime symbol:

$$\begin{aligned} X4'(i) &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \\ &= X3(i) \oplus X2(i) \oplus X1'(i) \oplus X0(i) \oplus X1(i) \oplus X1(i) \\ &= X3(i) \oplus X2(i) \oplus X1(i) \oplus X0(i) \oplus X1(i) \oplus X1'(i) \\ &= X4(i) \oplus X1(i) \oplus X1'(i) \end{aligned}$$

The preceding set of equations is derived as follows. The first line shows that a change in X1 will also affect the parity disk X4. In the second line, we add the terms [X1(i) X1(i)]. Because the exclusive-OR of any quantity with itself is 0, this does not affect the equation. However, it is a convenience that is used to create the third line, by reordering. Finally, the above equation is used to replace the first four terms by X4(i).

To calculate the new parity, the array management software must read the old user strip and the old parity strip. Then it can update these two strips with the new data and the newly calculated parity. Thus, each strip write involves two reads and two writes.

In the case of a larger size, I/O write that involves strips on all disk drives, parity is easily computed by calculation using only the new data bits. Thus, the parity drive can be updated in parallel with the data drives and there are no extra reads or writes.

2.7.6 RAID Level 5

RAID 5 is organized similarly to RAID 4. The difference is that RAID 5 distributes the parity strips across all disks. A typical allocation is a round-robin scheme, as illustrated in Figure 2.2 f. For an n -disk array, the parity strip is on a different disk for the first n stripes, and the pattern then repeats.

The distribution of parity strips across all drives avoids the potential I/O bottleneck of the single parity disk found in RAID 4. Further, RAID 5 has the characteristic that the loss of any one disk does not result in data loss.

2.7.7 RAID Level 6

RAID 6 was introduced in a subsequent paper by the Berkeley researchers. In the RAID 6 scheme, two different parity calculations are carried out and stored in separate blocks on different disks. Thus, a RAID 6 array whose user data require N disks consists of N + 2 disks.

Figure 2.2 g illustrates the scheme. P and Q are two different data check algorithms. One of the two is the exclusive-OR calculation used in RAID 4 and 5. But the other is an independent data check algorithm. This makes it possible to regenerate data even if two disks containing user data fail.

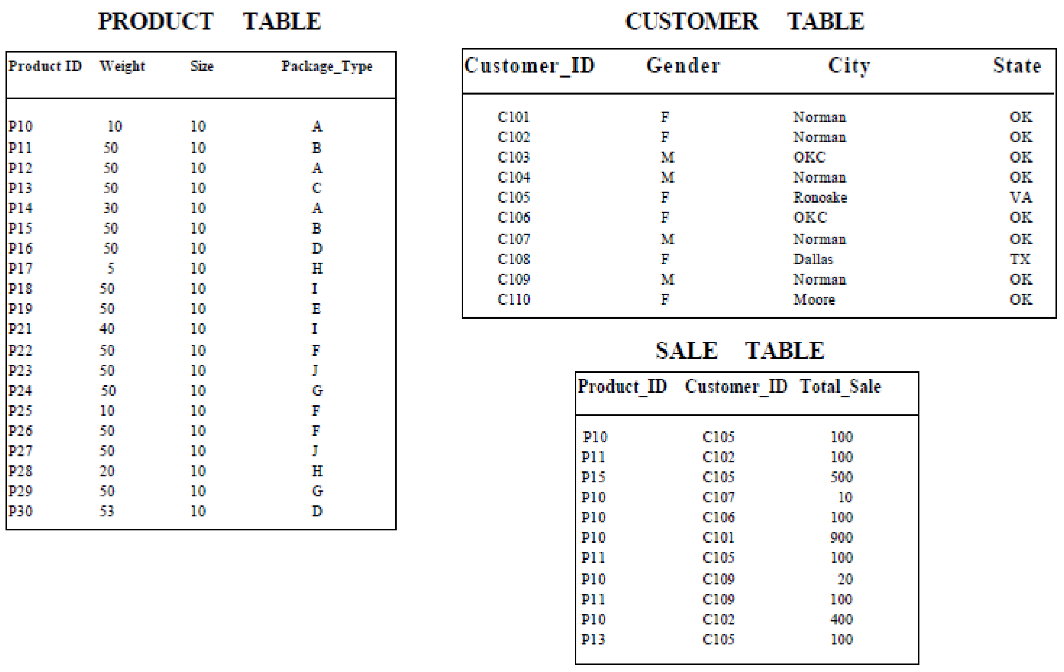
The advantage of RAID 6 is that it provides extremely high data availability. Three disks would have to fail within the MTTR (mean time to repair) interval to cause data to be lost. On the other hand, RAID 6 incurs a substantial write penalty, because each write affects two parity blocks. Performance benchmarks show that a RAID 6 controller can suffer more than a 30% drop in overall write performance compared with a RAID 5 implementation. RAID 5 and RAID 6 read performance is comparable.

2.8 Index And Types Of Indexes

Indexes are database objects associated with database tables and created to speed up access to data within the tables. Indexing techniques have already been in existence for decades for the OLTP relational database system but they cannot handle a large volume of data and complex and iterative queries that are common in OLAP applications.
In data warehouse systems, there are many indexing techniques. Each existing indexing technique is suitable for a particular situation. In this section we describe several indexing techniques being studied/used in both academic research and industrial applications. We will use the example in Figure 2.3 to explain the indexing techniques throughout the paper. The figure illustrates an example of a star schema with a central fact table called SALE and two dimension tables called PRODUCT and CUSTOMER.

2.8.1 The B$^+$Tree Index

The B$^+$Tree Index is the default index for most relational database systems. The topmost level of the index is called the root. The lowest level is called the leaf node. All other levels in between are called branches. Both the root and branch contain entries that point to the next level in the index.
Leaf nodes consisting of the index key and pointers pointing to the physical location (i.e., row ids) in which the corresponding records are stored. A B$^+$Tree Index for package_type of the PRODUCT table is shown in Figure 2.4.



PRODUCT TABLE

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

CUSTOMER TABLE

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

SALE TABLE

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

Figure 2.3 An Example of Product, Sale and Customer Table

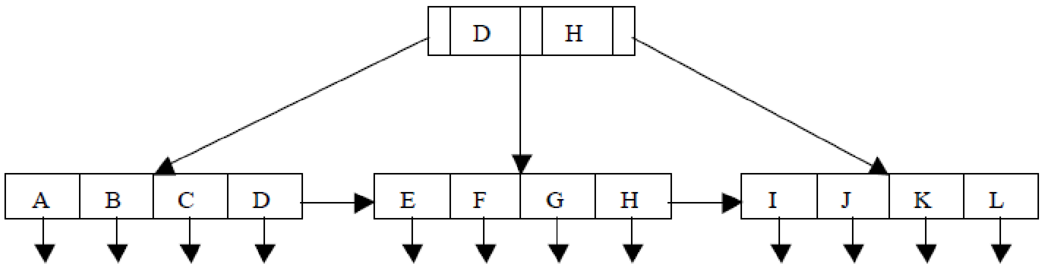


Figure 2.4 B$^+$ Tree index on Product Table

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

2.8.2 Projection Index

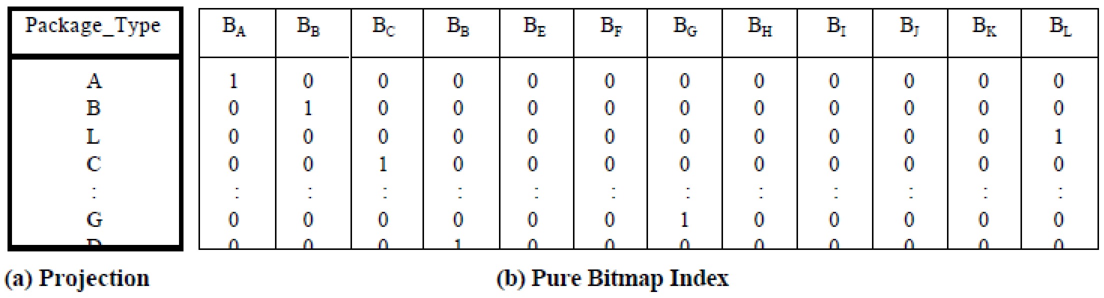
A Projection Index on an indexed column ***A*** in table ***T*** stores all values of ***A*** in the same order as they appear in ***T***. Each row of the Projection Index stores one value of ***A***. The row order of value ***x*** in the index is the same as the row order of value ***x*** in ***T***. Figure 2.5 shows the Projection Index on package_type of the PRODUCT table. Normally, the queries against a data warehouse retrieve only a few of the table's columns; so having the Projection Index on these columns reduces tremendously the cost of querying because a single I/O operation may bring more values into memory.

2.8.3 Bitmap Index

The bitmap representation is an alternate method of the row ids representation. It is simple to represent, and uses less space- and CPU-efficient than row ids when the number of distinct values of the indexed column is low. The indexes improve complex query performance by applying low-cost Boolean operations such as OR, AND, and NOT in the selection predicate on multiple indexes at one time to reduce search space before going to the primary source data. Many variations of the Bitmap Index (Pure Bitmap Index, Encoded Bitmap, etc.) have been introduced, aiming to reduce space requirement as well as improve query performance.

a) Pure Bitmap Index

Pure Bitmap Index was first introduced and implemented in the Model 204 DBMS. It consists of a collection of bitmap vectors each of which is created to represent each distinct value of the indexed column. A bit ***i*** in a bitmap vector, representing value ***x***, is set to 1 if the record ***i*** in the indexed table contains ***x***. Figure 2.5 shows an example of the Pure Bitmap Index on the package_type column of the PRODUCT table. The Pure Bitmap Index on this column is the collection of 12 bitmap vectors, says {BA, BB, BC, BD, BE, BF, BG, BH, BI, BJ, BK, and BL}, one for each package type. To answer a query, the bitmap vectors of the values specified in the predicate condition are read into memory. If there are more than one bitmap vectors read, a Boolean operation will be performed on them before accessing data. However, the sparsity problem occurs if the Pure Index is built on a high cardinality column which then requires more space and query processing time to build and answer a query. Most commercial data warehouse products (e.g., Oracle, Sybase, Informix, Red Brick, etc. implement the Pure Bitmap Index.



Package_Type
A
B
L
C
:
G
D

(a) Projection

B_A	B_B	B_C	B_B	B_E	B_F	B_G	B_H	B_I	B_J	B_K	B_L
1	0	0	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	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	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

(b) Pure Bitmap Index

Figure 2.5 Projection index and Bitmap index on Product Table

b) Encoded Bitmap Index

An Encoded Bitmap Index on column ***A*** of a table ***T*** consists of a set of bitmap vectors, a lookup table, and a set of retrieval Boolean functions. Each distinct value of column A is encoded using several bits each of which is stored in a bitmap vector. The lookup table stores the mapping between A and its encoded representation. IBM implements this index in DB2.

Comparing with the Pure Bitmap Index, the Encoded Bitmap Index improves space utilization and solves sparsity problems. The size of the Encoded Bitmap Index built on the high cardinality column is less than the Pure Bitmap Index. Having a well-defined encoding scheme, a Boolean operation can perform on the retrieval functions before retrieving the data and lead to a reduction of the number of bitmap vectors read. Its performance is degraded with equality queries since we have to search for all the bitmap vectors. The index needs to be rebuilt if we run out of bits to represent new values.

2.8.4 Join Index

A Join Index is built by translating restrictions on the column value of a dimension table (i.e., the gender column) to restrictions on a large fact table. The index is implemented using one of the two representations: row id or bitmap, depending on the cardinality of the indexed column. A bitmap representation, which is called Bitmap Join Index, is used with the low cardinality data while a row id representation is used with high cardinality. In DW, there are many join operations involved; so building Join Indexes on the joining columns improves query-processing time.

For example, Bitmap Join Indexes on the gender column in the SALE table can be built by using the gender column in the CUSTOMER table and the foreign key customer id in the SALES table. Note that the Sales table does not contain the gender column. The Bitmap Join Index for gender-equal to male is created by setting a bit corresponding to a row for customer_id whose gender is 'M' to 1 in the Sales Table. Otherwise, the bit is set to 0 as shown in Figure 2.6.

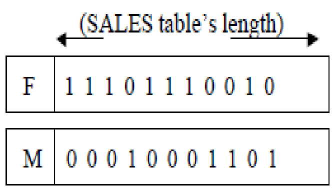


Figure 2.6 Bitmap join index on Sale Table

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

CHAPTER 3: PROBLEM STATEMENT

Most developers know that databases use data tables, stored procedures, views, and functions but indexes are often overlooked and misunderstood. Indexes are data structures that improve data retrieval from tables in a database. When data is stored on disk-based storage devices, it is stored as blocks of data. These blocks are accessed in their entirety, making them the atomic disk access operation. Disk blocks are structured in much the same way as linked lists; both contain a section for data, a pointer to the location of the next node (or block), and both need not be stored contiguously.

Because several records can only be sorted on one field, we can state that searching on a field that isn't sorted requires a Linear Search which requires N/2 block accesses (on average), where N is the number of blocks that the table spans. If that field is a non-key field (i.e. doesn't contain unique entries) then the entire tablespace must be searched at N block accesses.

Whereas with a sorted field, a Binary Search may be used, this has *log2 N* block accesses. Also since the data is sorted given a non-key field, the rest of the table doesn't need to be searched for duplicate values, once a higher value is found. Thus the performance increase is substantial.

Indexing is a way of sorting several records in multiple fields. Creating an index on a field in a table creates another data structure which holds the field value, and a pointer to the record it relates to. This index structure is then sorted, allowing Binary Searches to be performed on it.

Bitmap Indexing is one such indexing technique that uses bitmaps. Bitmap indexes use bit arrays (commonly called bitmaps) and answer queries by performing bitwise logical operations on these bitmaps. Bitmap indexes have a significant space and performance advantage over other structures for the query of low cardinality data. Bitmap indexes are also useful in data warehousing applications for joining a large fact tales to smaller dimension tables such as those arranged in a star schema.

A B+ tree is an n-ary tree with a variable but often a large number of children per node. A B+ tree consists of a root, internal nodes, and leaves. The root may be either a leaf or a node with two or more children. A B+ tree can be viewed as a B-tree in which each node contains only keys (not key-value pairs), and to which an additional level is added at the bottom with linked leaves. The primary value of a B+ tree is in storing data for efficient retrieval in a block-oriented storage context — in particular, filesystems. This is primarily because unlike binary search trees, B+ trees have very high fanout, which reduces the number of I/O operations required to find an element in the tree.

CHAPTER 4: PROPOSED METHODS

Bitmap indexes are widely used in data warehousing environments. The environments typically have large amounts of data and ad hoc queries, but a low level of concurrent DML transactions. For such applications, bitmap indexing provides:

- Reduced response time for large classes of ad hoc queries
- Reduced storage requirements compared to other indexing techniques
- Dramatic performance gains even on hardware with a relatively small number of CPUs or a small amount of memory
- Efficient maintenance during parallel DML and loads

4.1 Bitmap index

Bitmap indexes are often useful in many situations where B-tree indexes are not optimal. Namely, for columns that have a large number of duplicate values. For example, a Size column that contains only five distinct values (tiny, small, medium, large, grand). Fully indexing a large table with a traditional B-tree index can be prohibitively expensive in terms of space because the indexes can be several times larger than the data in the table. Bitmap indexes are typically only a fraction of the size of the indexed data in the table.

An index provides pointers to the rows in a table that contain a given key value. A regular index stores a list of row IDs for each key corresponding to the rows with that key value. In a bitmap index, a bitmap for each key-value replaces a list of rowids.

Each bit in the bitmap corresponds to a possible rowid, and if the bit is set, it means that the row with the corresponding rowid contains the key value. A mapping function converts the bit position to an actual row ID so that the bitmap index provides the same functionality as a regular index. If the number of different key values is small, bitmap indexes save space.

Bitmap indexes are most effective for queries that contain multiple conditions in the WHERE clause. Rows that satisfy some, but not all, conditions are filtered out before the table itself is accessed. This improves response time, often dramatically.

4.1.1 Candidates for Bitmap Indexes

Bitmap indexes are most advantageous whenever the cardinality of the index is less than one percent or lowly-selective. This criterion is nearly the opposite of the guideline for B-Tree indexes.

Bitmap indexing has to be used in the following scenarios:

- A query constrains multiple columns that have few distinct values in their domains (a large number of duplicate values).
- A large number of rows satisfy the constraints on these columns.
- Bitmap indexes have been created on some or all of these columns.
- The referenced table contains a large number of rows.

Given this kind of scenario, the server can evaluate the constraints by ANDing the bitmaps and potentially eliminate a large number of rows without ever accessing a row in the table.

The *Oracle* database server can also generate query execution plans to join a fact table to its dimensions using its star transformation algorithm when bitmap indexes exist on the fact table foreign key reference columns. When this execution plan is the least costly, the server joins the tables using the bitmap indexes.

Note: Bitmap indexes should be used only for static tables and are not suited for highly volatile tables in online transaction processing systems.

4.1.2 Cardinality

The advantages of using bitmap indexes are greatest for columns in which the ratio of the number of distinct values to the number of rows in the table is under 1%. We refer to this ratio as the **degree of cardinality**. A gender column, which has only two distinct values (male and female), is ideal for a bitmap index. However, data warehouse administrators also build bitmap indexes on columns with higher cardinalities.

For example, on a table with one million rows, a column with 10,000 distinct values is a candidate for a bitmap index. A bitmap index on this column can outperform a B-tree index, particularly when this column is often queried in conjunction with other indexed columns. In fact, in a typical data warehouse environments, a bitmap index can be considered for any non-unique column.

B-tree indexes are most effective for high-cardinality data: that is, for data with many possible values, such as customer_name or phone_number. In a data warehouse, B-tree indexes should be used only for unique columns or other columns with very high cardinalities (that is, almost unique columns). The majority of indexes in a data warehouse should be bitmap indexes.

In ad hoc queries and similar situations, bitmap indexes can dramatically improve query performance. AND and OR conditions in the WHERE clause of a query can be resolved quickly by performing the corresponding Boolean operations directly on the bitmaps before converting the resulting bitmap to row IDs. If the resulting number of rows is small, the query can be answered quickly without resorting to a full table scan.

Assume that the column for color has three distinct colors, namely, white, almond, and black. Construct a bitmap using these three distinct values. Each entry in the bitmap contains three bits. Let us say the first bit refers to white, the second to almond, and the third to black. If a product is white, the bitmap entry for that product consists of three bits, where the first bit is set to 1, the second bit is set to 0, and the third bit is set to 0. If a product is an almond in color, the bitmap entry for that product consists of three bits, where the first bit is set to 0, the second bit is set to 1, and the third bit is set to 0. Now study the bitmapped index example shown in Figure 4.1. The figure presents an extract of the sales table and bitmapped indexes for the three different columns. Notice how each entry in an index contains the ordered bits to represent the distinct values in the column. An entry is created for each row in the base table. Each entry carries the address of the base table row.
How do the bitmapped indexes work to retrieve the requested rows? Consider a query against the sales table in the above example*:*

Select the rows from the Sales table
Where Product is "Washer" and
Color is "Almond" and Division is "East" or "South"

Figure 4.2 illustrates how Boolean logic is applied to find the result set based on the bitmapped indexes shown in Figure 4.1.

As you may observe, bitmapped indexes support queries using low-selectivity columns. The strength of this technique rests on its effectiveness when using predicates on low-selectivity columns in queries. Bitmapped indexes take significantly less space than B-Tree indexes for low-selectivity columns. In a data warehouse, many data accesses are based on low-selectivity columns. Also, analysis using "what-if" scenarios require queries involving several predicates. You will find that bitmapped indexes are more suitable for a data warehouse environment than for an OLTP system.

Extract of Sales Data

Address or Rowid	Date	Product	Color	Region	Sale ($)
00001BFE.0012.0111	15-Nov-00	Dishwasher	White	East	300
00001BFE.0013.0114	15-Nov-00	Dryer	Almond	West	450
00001BFF.0012.0115	16-Nov-00	Dishwasher	Almond	West	350
00001BFF.0012.0138	16-Nov-00	Washer	Black	North	550
00001BFF.0012.0145	17-Nov-00	Washer	White	South	500
00001BFF.0012.0157	17-Nov-00	Dryer	White	East	400
00001BFF.0014.0165	17-Nov-00	Washer	Almond	South	575

Bitmapped Index for Product Column
Ordered bits: Washer, Dryer, Dishwasher

Address or Rowid	Bitmap
00001BFE.0012.0111	001
00001BFE.0013.0114	010
00001BFF.0012.0115	001
00001BFF.0012.0138	100
00001BFF.0012.0145	100
00001BFF.0012.0157	010
00001BFF.0014.0165	100

Bitmapped Index for Color Column
Ordered bits: White, Almond, Black

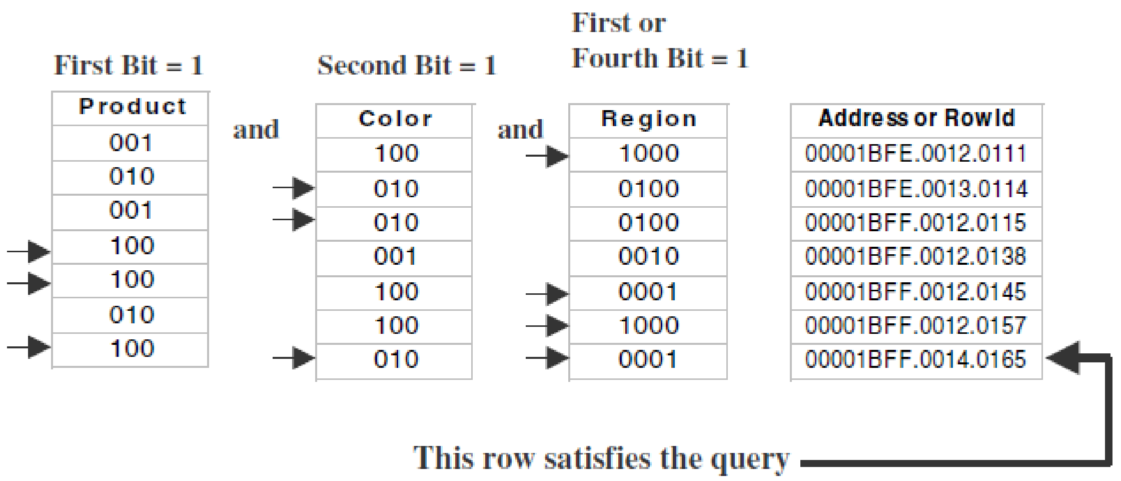
Address or Rowid	Bitmap
00001BFE.0012.0111	100
00001BFE.0013.0114	010
00001BFF.0012.0115	010
00001BFF.0012.0138	001
00001BFF.0012.0145	100
00001BFF.0012.0157	100
00001BFF.0014.0165	010

Bitmapped Index for Region Column
Ordered bits: East, West, North, South

Address or Rowid	Bitmap
00001BFE.0012.0111	1000
00001BFE.0013.0114	0100
00001BFF.0012.0115	0100
00001BFF.0012.0138	0010
00001BFF.0012.0145	0001
00001BFF.0012.0157	1000
00001BFF.0014.0165	0001

Figure 4.1 Bitmapped index example

Select the rows from Sales
Where Product is Washer, and
Color is Almond, and
The division is East or South.



Product	Color	Region	Address or RowId
001	100	→ 1000	00001BFE.0012.0111
010	→ 010	0100	00001BFE.0013.0114
001	→ 010	0100	00001BFF.0012.0115
→ 100	001	0010	00001BFF.0012.0138
→ 100	100	→ 0001	00001BFF.0012.0145
010	100	→ 1000	00001BFF.0012.0157
→ 100	→ 010	→ 0001	← 00001BFF.0014.0165

Figure 4.2 Bitmapped indexes: data retrieval

When Bitmap Indexes Should Not Be Created

Bitmap indexes should not be created in the following cases:

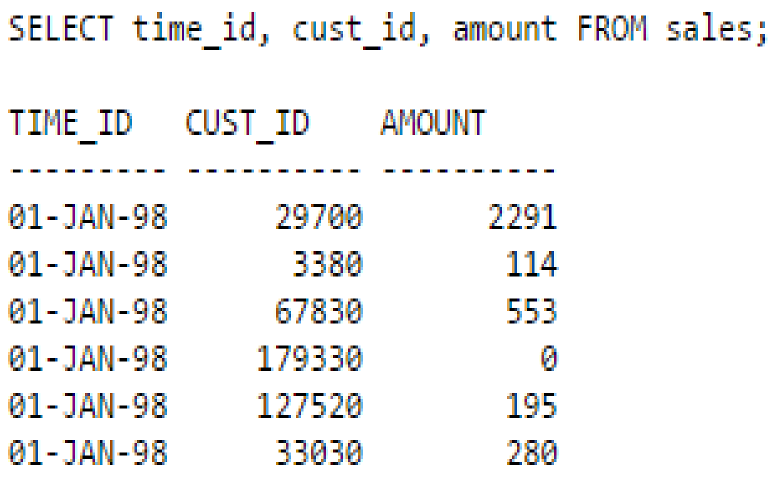
- A column with a UNIQUE constraint
- A column that mostly contains distinct values
- Where the table is frequently updated or loaded with new data

4.1.3 Bitmap Join Indexes

In addition to a bitmap index on a single table, you can create a bitmap join index, which is a bitmap index for the join of two or more tables. A bitmap join index is a space-efficient way of reducing the volume of data that must be joined by performing restrictions in advance. For each value in a column of a table, a bitmap joins index stores the row IDs of corresponding rows in one or more other tables. In a data warehousing environment, the join condition is an equi-inner join between the primary key column or columns of the dimension tables and the foreign key column or columns in the fact table.

Bitmap join indexes are much more efficient in storage than materialized join views, an alternative for materializing joins in advance. This is because the materialized join views do not compress the row IDs of the fact tables.

Bitmap Join Index: Example1



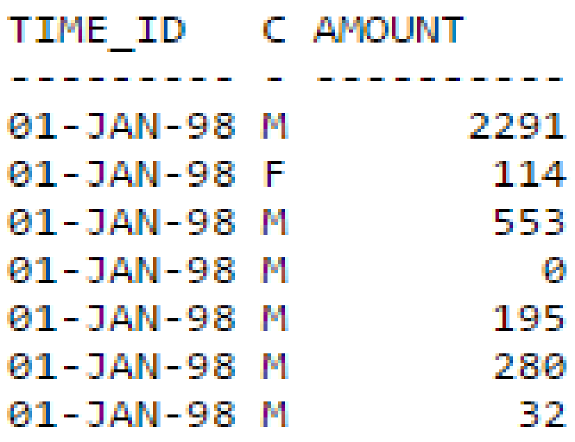
```
SELECT time_id, cust_id, amount FROM sales;

TIME_ID    CUST_ID     AMOUNT
--------- ---------- ----------
01-JAN-98      29700       2291
01-JAN-98       3380        114
01-JAN-98      67830        553
01-JAN-98     179330          0
01-JAN-98     127520        195
01-JAN-98      33030        280
```

Figure 4.3 Bitmap join index example1

```
CREATE BITMAP INDEX sales_cust_gender_bjix
          ON sales (customers.cust_gender)
                    FROM sales, customers
                              WHERE sales.cust_id = customers.cust_id LOCAL;
```

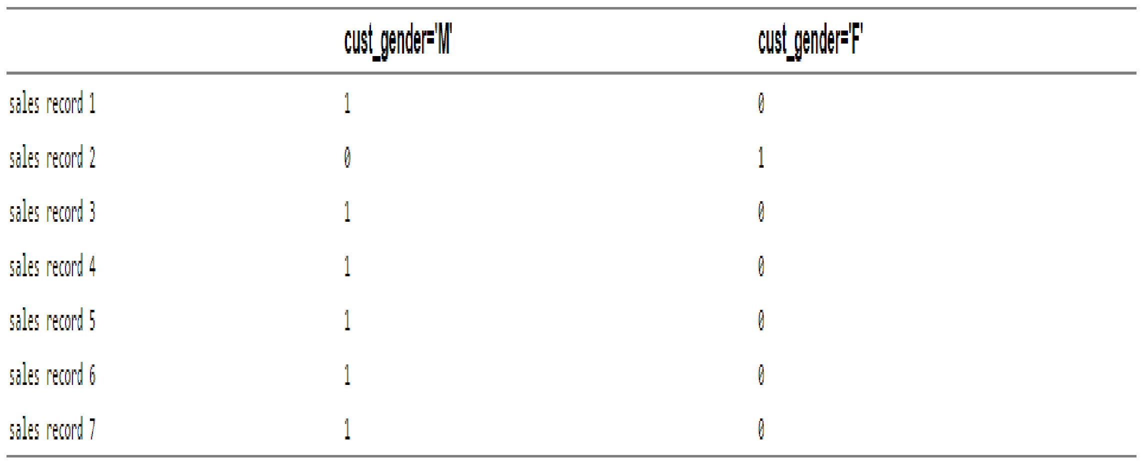
The following query shows how to use this bitmap join index and illustrates its bitmap pattern:

```
          SELECT sales.time_id, customers.cust_gender, sales.amount
                    FROM sales, customers
                              WHERE sales.cust_id = customers.cust_id;
```



```
TIME_ID   C AMOUNT
--------- - ----------
01-JAN-98 M       2291
01-JAN-98 F        114
01-JAN-98 M        553
01-JAN-98 M          0
01-JAN-98 M        195
01-JAN-98 M        280
01-JAN-98 M         32
```

Table 4.1 Sample Bitmap Join Index



	cust_gender='M'	cust_gender='F'
sales record 1	1	0
sales record 2	0	1
sales record 3	1	0
sales record 4	1	0
sales record 5	1	0
sales record 6	1	0
sales record 7	1	0

4.1.4 Bitmap Join Index Restrictions

Join results must be stored, therefore, bitmap join indexes have the following restrictions:

- Parallel DML is currently only supported on the fact table. Parallel DML on one of the participating dimension tables will mark the index as unusable.
- Only one table can be updated concurrently by different transactions when using the bitmap join index.
- No table can appear twice in the join.
- You cannot create a bitmap join index on an index-organized table or a temporary table.
- The columns in the index must all be columns of the dimension tables.
- The dimension table joins columns must be either primary key columns or have unique constraints.
- If a dimension table has a composite primary key, each column in the primary key must be part of the join.

4.2 B$^+$Tree Index

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by a reorganization of the file, frequent reorganizations are undesirable.
The B$^+$Tree index structure is the most widely used of several index structures that maintain their efficiency despite the insertion and deletion of data. A B$^+$Tree index takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. Each non-leaf node in the tree has between $n/2$ and n children, where n is fixed for a particular tree.
We shall see that the B$^+$Tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B$^+$Tree structure.
A B$^+$Tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Figure 4.4 shows a typical node of a B$^+$Tree. It contains up to $n - 1$ search-key values $K1, K2, \ldots, Kn-1$, and n pointers $P1, P2, \ldots, Pn$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $Ki < Kj$.
We consider first the structure of the leaf nodes. For $i = 1, 2, \ldots, n-1$, pointer
Pi points to a file record with search-key value Ki. Pointer Pn has a special purpose that we shall discuss shortly.

The figure shows one leaf node of a B$^+$Tree for the *instructor* file, in which we have chosen
n to be 4, and the search key is the *name*.
Now that we have seen the structure of a leaf node, let us consider how search-key values are assigned to particular nodes. Each leaf can hold up to $n - 1$ values. We allow leaf nodes to contain as few as $(n - 1)/2$ values. With $n = 4$ in our example B$^+$Tree, each leaf must contain at least 2 values, and at most 3 values.
The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf.
Specifically, if Li and $L\,j$ are leaf nodes and $i < j$, then every search-key value in Li is less than or equal to every search-key value in $L\,j$. If the B$^+$Tree index is used as a dense index (as is usually the case) every search-key value must appear in some leaf node.
Now we can explain the use of the pointer *Pn*. Since there is a linear order on the leaves based on the search-key values that they contain, we use *Pn* to chain

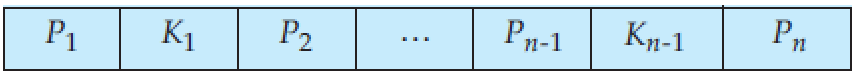
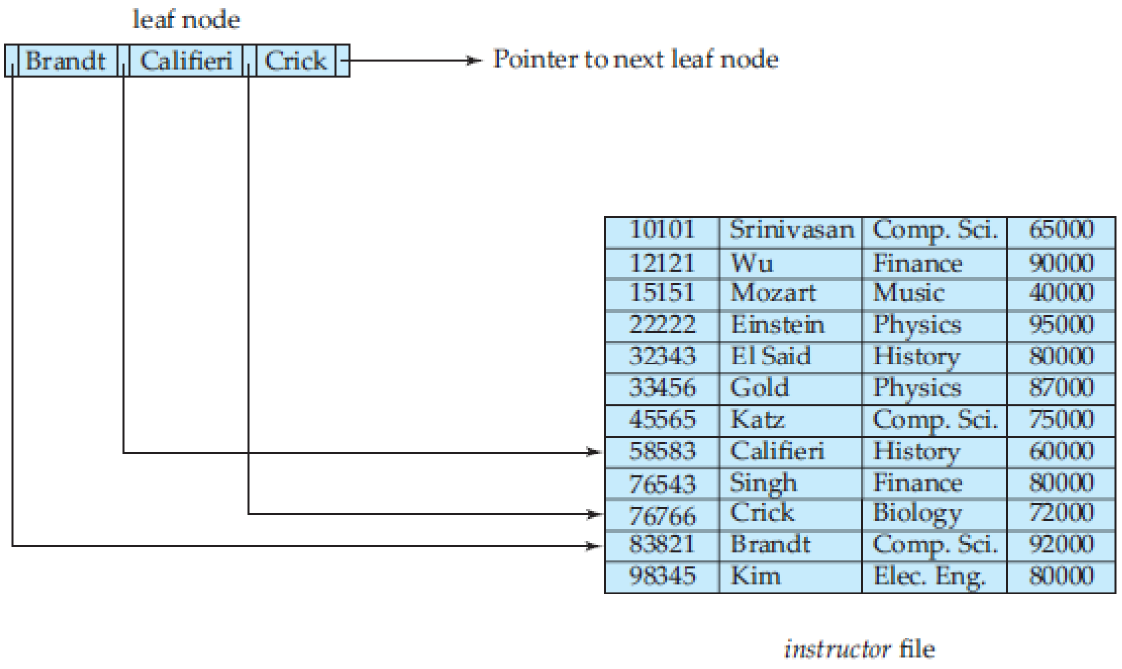


Figure 4.4 A sample B$^+$Tree Node



10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	80000
33456	Gold	Physics	87000
45565	Katz	Comp. Sci.	75000
58583	Califieri	History	60000
76543	Singh	Finance	80000
76766	Crick	Biology	72000
83821	Brandt	Comp. Sci.	92000
98345	Kim	Elec. Eng.	80000

Figure 4.5 A leaf node for instructor B$^+$Tree Index(n = 4)

together with the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file.
The non-leaf nodes of the B$^+$Tree form a multilevel (sparse) index on the leaf nodes. The structure of non-leaf nodes is the same as that for leaf nodes, except that all

pointers are pointers to tree nodes. A non-leaf node may hold up to *n* pointers and *must* hold at least *n/*2 pointers. The number of pointers in a node is called the *fanout* of the node. Non-leaf nodes are also referred to as internal nodes.
Let us consider a node containing *m* pointers ($m \leq n$). For $i = 2, 3, \ldots, m - 1$,
pointer *Pi* points to the subtree that contains search-key values less than *Ki* and greater than or equal to *Ki* −1. Pointer *Pm* points to the part of the subtree that
contains those key values greater than or equal to *Km*−1, and pointer *P*1 points to the part of the subtree that contains those search-key values less than *K*1.
Unlike other non-leaf nodes, the root node can hold fewer than _*n/*2_ pointers; however, it must hold at least two pointers, unless the tree consists of only one node. It is always possible to construct a B^+Tree, for any *n*, that satisfies the preceding requirements.
Figure 4.5 shows a complete B^+Tree for the *instructor* file (with $n = 4$). We have shown instructor names abbreviated to 3 characters to depict the tree clearly; in reality, the tree nodes would contain the full names. We have also omitted null pointers for simplicity; any pointer field in the figure that does not have an arrow is understood to have a null value.

4.2.1 B^+tree Insertion

Since all objects in the B^+tree are located at the leaf level, the insertion algorithm of the B^+tree is easier than that in the B-tree. We follow the exact-match query to and the leaf node which should contain the object if it were in the tree. Then we insert the object into the leaf node.
What needs to be taken care of is when the leaf node overflows and is split into two. In this case, a key and a child pointer are inserted into the parent node. This may in turn cause the parent node to overflow, and so on. In the worst case, all nodes along the insertion path are split. If the root node splits into two, the height of the tree increases by one.

4.2.2 Handling Duplicate Keys in B^+Trees

There are several different ways of handling duplicate keys. One way is to use an unmodified insert algorithm that allows duplicate keys in blocks but is otherwise unchanged. The issue with a structure such as this is the search algorithm must be modified to take into account several corner cases that arise. For instance one of the invariants of a B^+Tree may be violated in this structure. Specifically, if there are many duplicate keys, a copy of one of the keys may be in a non-leaf block. However, the key may appear in blocks that appear logically before the block which is pointed at by the key in the internal block. Thus the search algorithm must be modified to look in the previous blocks to the one suggested by the unmodified search algorithm. This will slow down the common case for search.

There is another issue with this straight forward implementation, if there are many duplicate keys in the index, the index size may be taller than necessary. Consider a situation where for each unique key there are perhaps hundreds of duplicates, the index size will be proportional to the total number of keys in the main file, however, you only need to index an index on the unique keys. One of the files indexed in our program will be indexing has such characteristics to its data. It indexes strings (as the keys) with associated instances where those strings show up in our documents. There can be hundreds to thousands of instances of each unique string.

Therefore the approach I took was to store only the unique keys in the index and have the duplicates captured in overflow blocks in the main file. An example of such a tree can be seen in figure 4.6. Consider key 6; there are 5 instances of this key in the tree. The tree is order-3, indicating the keys cannot all fit in one block. To handle this situation an overflow block is chained to the block which is indexed by the tree structure. The overflow block then points to the next relevant block in the tree.

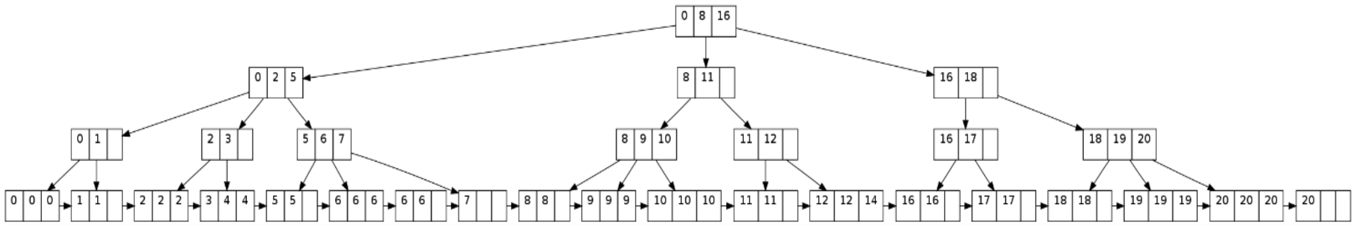


Figure 4.6 B⁺Tree Duplicates Insertion Visualization

4.2.3 Deletion In B⁺Tree

Descend to the leaf where the key exists.

Remove the required key and associated references from the node.

If the node still has enough keys and references to satisfy the invariants, stop.

If the node has too few keys to satisfy the invariants, but it's next oldest or next youngest sibling at the same level has more than necessary, distribute the keys between this node and the neighbor. Repair the keys in the level above to represent that these nodes now have a different “split point” between them; this involves simply changing a key in the levels above, without deletion or insertion.

If the node has too few keys to satisfy the invariant, and the next oldest or next youngest sibling is at the minimum for the invariant, then merge the node with its sibling; if the node is a non-leaf, we will need to incorporate the “split key” from the parent into our merging. In either case, we will need to repeat the removal algorithm on the parent node to remove the “split key” that previously separated these merged nodes — unless the parent is the root and we are removing the final key from the root, in which case the merged node becomes the new root (and the tree has become one level shorter than before).

CHAPTER 5: IMPLEMENTATION

A database index is a data structure that improves the speed of data retrieval operations on a database table at the cost of additional writes and storage space to maintain the index data structure. Indexes are used to quickly locate data without having to search every row in a database table every time a database table is accessed.

Indexes can be created using one or more columns of a database table, providing the basis for both rapid random lookups and efficient access of ordered records.

Bitmap indexes use bit arrays (commonly called bitmaps) and answer queries by performing bitwise logical operations on these bitmaps. Bitmap indexes have a significant space and performance advantage over other structures for the query of such data. To maintain and retrieve such a huge amount of data efficiently at the same time minimizing the retrieval time, we use Bitmap indexing. Many algorithms were previously defined for such retrieval but they have their postulates and limitations. Our research includes studying the currently used algorithms and try to come up with a refined solution.

Coming to the B^+Trees, the leaves (the bottom-most index blocks) of the B+ tree are often linked to one another in a linked list; this makes range queries or an (ordered) iteration through the blocks simpler and more efficient (though the aforementioned upper bound can be achieved even without this addition). This does not substantially increase space consumption or maintenance on the tree. This illustrates one of the significant advantages of a B^+Tree over a B-tree; in a B-tree, since not all keys are present in the leaves, such an ordered linked list cannot be constructed. A B^+Tree is thus particularly useful as a database system index, where the data typically resides on disk, as it allows the B^+Tree to provide an efficient structure for housing the data itself.

If nodes of the B+ tree are organized as arrays of elements, then it may take a considerable time to insert or delete an element as half of the array will need to be shifted on average. To overcome this problem, elements inside a node can be organized in a binary tree or a B+ tree instead of an array.

Data warehousing involves a huge amount of data. To implement indexing on data of this order, we have supporting technologies like JAVA, MYSQL. JAVA is used to build the front end and for testing. MYSQL is a back-end tool used for storage and retrieval.

Tools Used: Netbeans, MySQL Workbench

5.1 Bitmap Indexing On Files

For implementing Bitmap indexing on files, we considered a large table consisting of 500000 odd records. This hash separated file is around 50 MB in size and with each record 100 bytes size. We have given this file path as input to the Bitmap algorithm where each line is parsed and bitmap index file is generated by creating indexing on **department** and **rating** attributes making it a multidimensional bitmap table. Because bitmap indexing works best on low cardinality data, we have chosen **department** and **rating** for applying bitmap indexing. By creating bitmap indexing on this file, the file gets compressed from 50 MB to 12 MB.

5.1.1 Front End Design

The User Interface was built using JAVA to filter the records based on department and rating which is provided by the user.

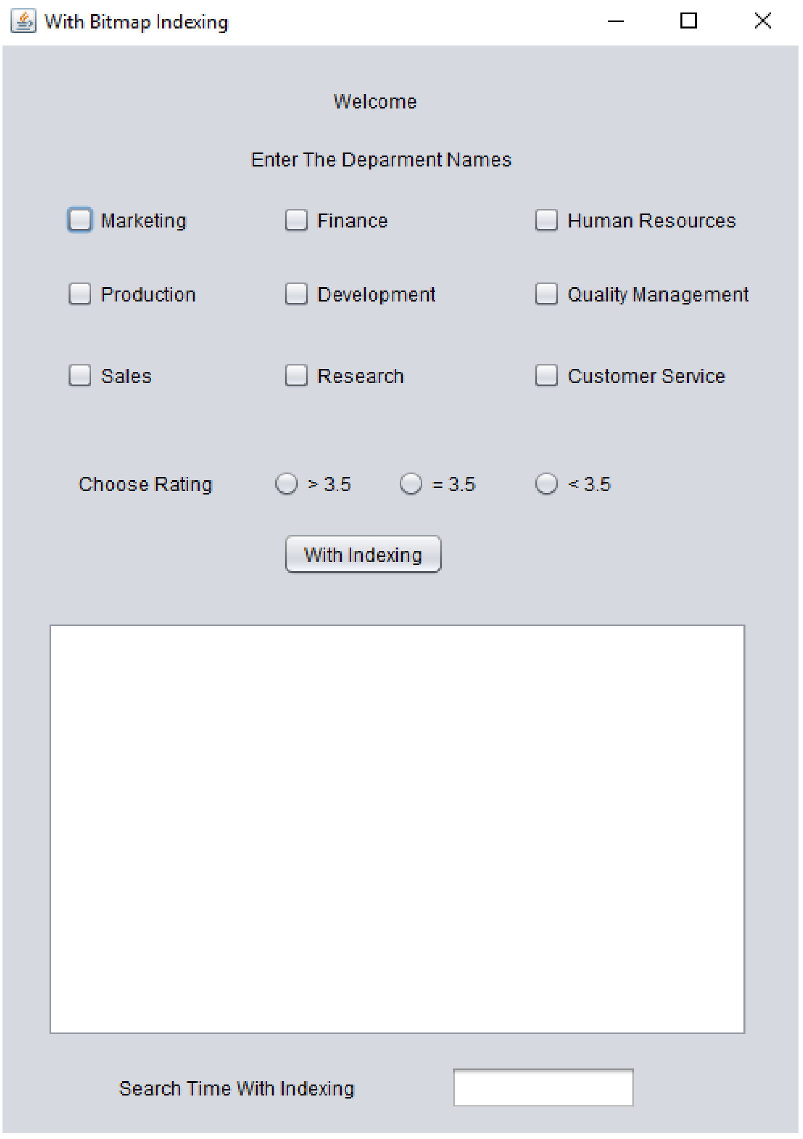


Figure 5.1 User Interface

The screen gives the user the option of selecting the departments (one or multiple) from the given set of departments and selecting the rating of the employee. The employee details will get displayed in the text area. The retrieval time is also displayed and comparison is done on these values.

5.2 Bitmap Indexing- Implementation

Each line is parsed till the end of the file is encountered. The whole record is captured in an ArrayList, the department under which the value is "1", a check is made to see if this department name matches the department name entered by the user. The same is done on the rating of the employee as well. If both these criteria are met, the record gets displayed in the text area of the UI.

5.3 B$^+$Tree Indexing- Implementation

Our hash-separated file contains nine unique departments on which the B$^+$Tree indexing algorithm is applied.

5.3.1 Insertion into B$^+$Tree

The algorithm goes as follows:

1. We initially specify the degree and order of the B$^+$Tree based on which filling into nodes is done.
2. If the node has space, insert the department name as Key into the non-leaf node.
3. The leaf node contains the data corresponding to that department.
4. If the node is already full, split it into two nodes, distributing the keys evenly between the two nodes. If the node is a leaf, take a copy of the minimum value in the second of these two nodes and repeat this insertion algorithm to insert it into the parent node. If the node is a non-leaf, exclude the middle value during the split and repeat this insertion algorithm to insert this excluded value into the parent node.

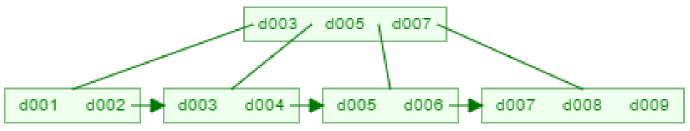


Figure 5.2 B$^+$Tree Visualization

5.3.2 Searching the B^+Tree

Searching is similar to searching a binary search tree. Starting at the root, the tree is recursively traversed from top to bottom. At each level, the search chooses the child pointer (subtree) whose separation values are on either side of the search value. Binary search is typically (but not necessarily) used within nodes to find the separation values and child tree of interest.

5.4 Without Indexing- Implementation

Implementing the sequential search algorithm involves parsing the hash separated file line by line. Here a string comparison is done between the department name of that record and the **department** name entered by the user. The same rule follows even for the **rating** attribute. If both the criterion are met, that record gets displayed in the text area of the UI.

Since this is a comparison between two strings, the program has to convert these strings into bits of 1’s and 0’s before interacting with the hardware which is a considerable overhead degrading the performance.

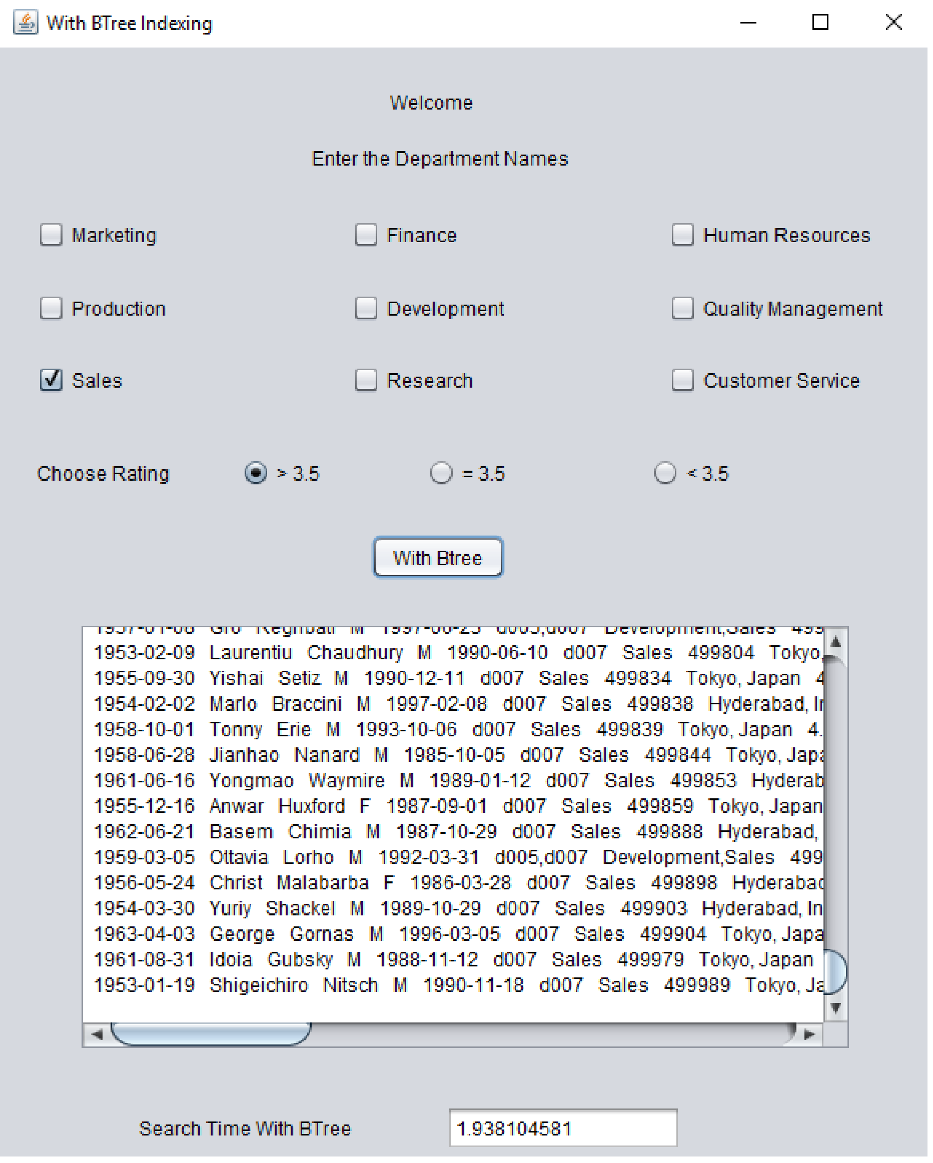
CHAPTER 6: RESULTS AND PERFORMANCE EVALUATION

In this chapter, we evaluate the performance of B$^+$Tree indexing, Bitmap indexing, and sequential search algorithms by comparing their retrieval time against a set of departments as shown in the figures below.

These timing measurements directly reflect the performance of indexing methods. A summary of all the timing measurements is indicated using graphs below.

6.1 With B$^+$Tree Indexing

The User Interface consists of nine departments as shown below. The user can select any department or a combination of departments along with the rating. Based on the parameters selected by the user, the B$^+$Tree algorithm is run and retrieval time is displayed.



With BTree Indexing

Welcome

Enter the Department Names

- [ ] Marketing
- [ ] Finance
- [ ] Human Resources
- [ ] Production
- [ ] Development
- [ ] Quality Management
- [✓ Sales]
- [ ] Research
- [ ] Customer Service

Choose Rating: [✓ > 3.5] = 3.5 < 3.5

With Btree

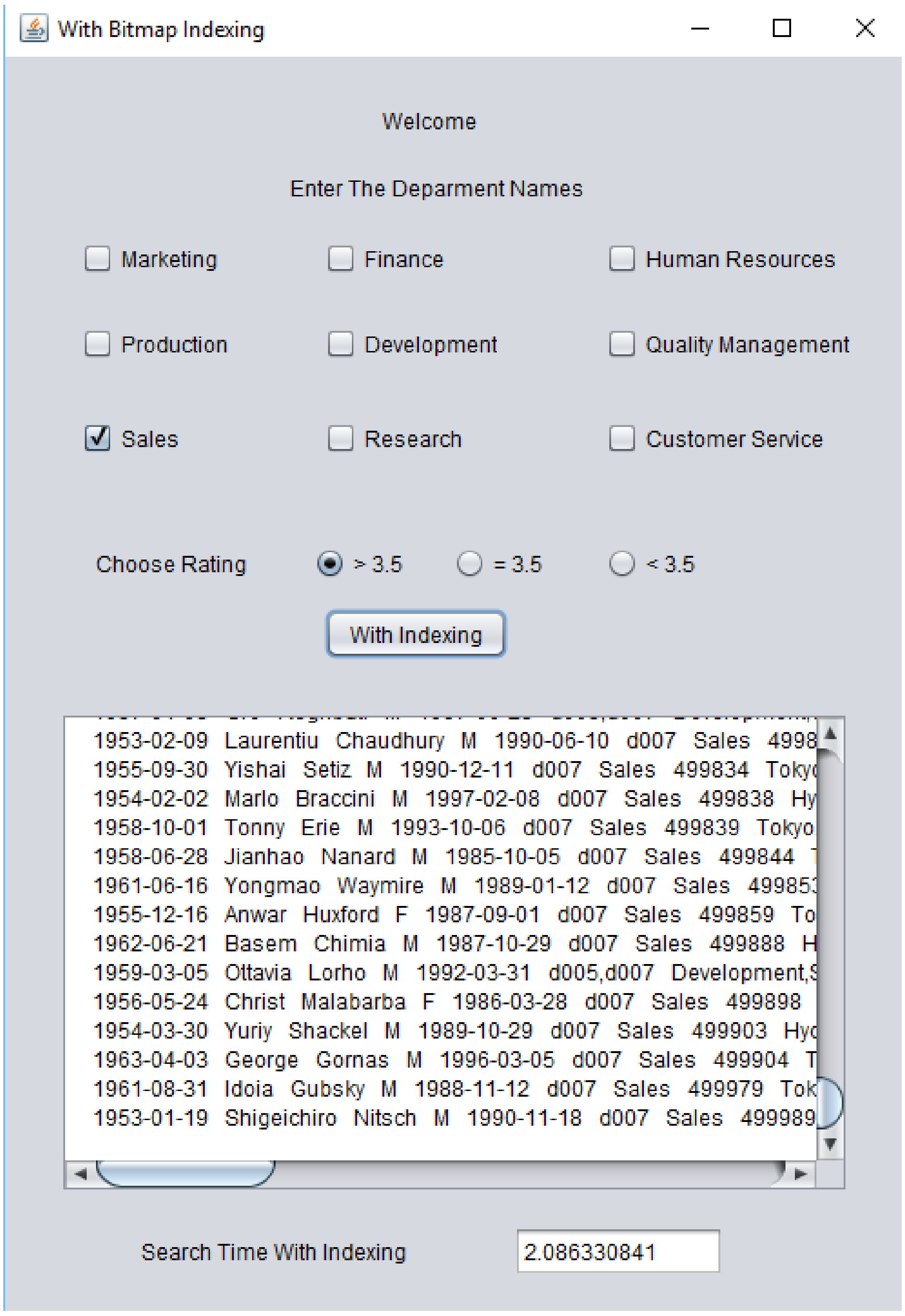
```
1957-01-08  Gro  Reghbati  M  1997-06-23  d005,d007  Development,Sales  499
1953-02-09  Laurentiu  Chaudhury  M  1990-06-10  d007  Sales  499804  Tokyo,
1955-09-30  Yishai  Setiz  M  1990-12-11  d007  Sales  499834  Tokyo, Japan  4
1954-02-02  Marlo  Braccini  M  1997-02-08  d007  Sales  499838  Hyderabad, Ir
1958-10-01  Tonny  Erie  M  1993-10-06  d007  Sales  499839  Tokyo, Japan  4.
1958-06-28  Jianhao  Nanard  M  1985-10-05  d007  Sales  499844  Tokyo, Japa
1961-06-16  Yongmao  Waymire  M  1989-01-12  d007  Sales  499853  Hyderab
1955-12-16  Anwar  Huxford  F  1987-09-01  d007  Sales  499859  Tokyo, Japan
1962-06-21  Basem  Chimia  M  1987-10-29  d007  Sales  499888  Hyderabad,
1959-03-05  Ottavia  Lorho  M  1992-03-31  d005,d007  Development,Sales  499
1956-05-24  Christ  Malabarba  F  1986-03-28  d007  Sales  499898  Hyderabac
1954-03-30  Yuriy  Shackel  M  1989-10-29  d007  Sales  499903  Hyderabad, In
1963-04-03  George  Gornas  M  1996-03-05  d007  Sales  499904  Tokyo, Japa
1961-08-31  Idoia  Gubsky  M  1988-11-12  d007  Sales  499979  Tokyo, Japan
1953-01-19  Shigeichiro  Nitsch  M  1990-11-18  d007  Sales  499989  Tokyo, Ja
```

Search Time With BTree: 1.938104581

Figure 6.1 With B$^+$Tree Indexing

6.2 With Bitmap Indexing

The User Interface consists of nine departments as shown below. The user can select any department or a combination of departments along with the rating. Based on the parameters selected by the user, the Bitmap indexing algorithm is run and retrieval time is displayed.



With Bitmap Indexing

Welcome

Enter The Deparment Names

[ ] Marketing [ ] Finance [ ] Human Resources

[ ] Production [ ] Development [ ] Quality Management

[✓ Sales] [ ] Research [ ] Customer Service

Choose Rating [✓ > 3.5] = 3.5 < 3.5

With Indexing

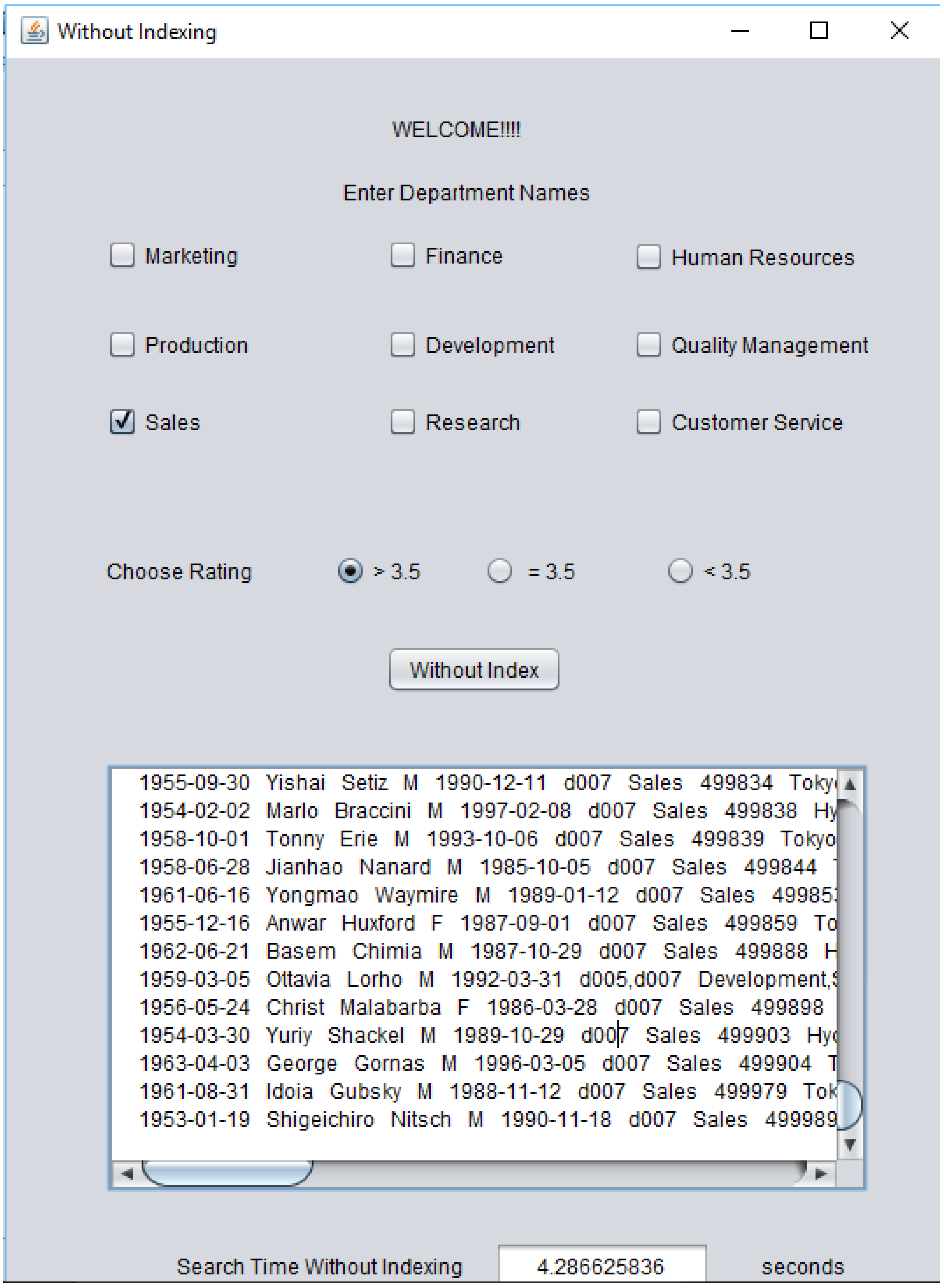
```
1953-02-09  Laurentiu  Chaudhury  M  1990-06-10  d007  Sales  4998
1955-09-30  Yishai  Setiz  M  1990-12-11  d007  Sales  499834  Toky
1954-02-02  Marlo  Braccini  M  1997-02-08  d007  Sales  499838  Hy
1958-10-01  Tonny  Erie  M  1993-10-06  d007  Sales  499839  Tokyo
1958-06-28  Jianhao  Nanard  M  1985-10-05  d007  Sales  499844  T
1961-06-16  Yongmao  Waymire  M  1989-01-12  d007  Sales  49985
1955-12-16  Anwar  Huxford  F  1987-09-01  d007  Sales  499859  To
1962-06-21  Basem  Chimia  M  1987-10-29  d007  Sales  499888  H
1959-03-05  Ottavia  Lorho  M  1992-03-31  d005,d007  Development,
1956-05-24  Christ  Malabarba  F  1986-03-28  d007  Sales  499898
1954-03-30  Yuriy  Shackel  M  1989-10-29  d007  Sales  499903  Hy
1963-04-03  George  Gornas  M  1996-03-05  d007  Sales  499904  T
1961-08-31  Idoia  Gubsky  M  1988-11-12  d007  Sales  499979  Tok
1953-01-19  Shigeichiro  Nitsch  M  1990-11-18  d007  Sales  499989
```

Search Time With Indexing 2.086330841

Figure 6.2 With Bitmap Indexing

6.3 Without Indexing

The User Interface consists of nine departments as shown below. The user can select any department or a combination of departments along with the rating. Based on the parameters selected by the user, the sequential search algorithm is run and retrieval time is displayed.



```
1955-09-30  Yishai  Setiz  M  1990-12-11  d007  Sales  499834  Toky
1954-02-02  Marlo  Braccini  M  1997-02-08  d007  Sales  499838  Hy
1958-10-01  Tonny  Erie  M  1993-10-06  d007  Sales  499839  Tokyo
1958-06-28  Jianhao  Nanard  M  1985-10-05  d007  Sales  499844  T
1961-06-16  Yongmao  Waymire  M  1989-01-12  d007  Sales  49985
1955-12-16  Anwar  Huxford  F  1987-09-01  d007  Sales  499859  To
1962-06-21  Basem  Chimia  M  1987-10-29  d007  Sales  499888  H
1959-03-05  Ottavia  Lorho  M  1992-03-31  d005,d007  Development,S
1956-05-24  Christ  Malabarba  F  1986-03-28  d007  Sales  499898  
1954-03-30  Yuriy  Shackel  M  1989-10-29  d007  Sales  499903  Hy
1963-04-03  George  Gornas  M  1996-03-05  d007  Sales  499904  T
1961-08-31  Idoia  Gubsky  M  1988-11-12  d007  Sales  499979  Tok
1953-01-19  Shigeichiro  Nitsch  M  1990-11-18  d007  Sales  499989
```

Search Time Without Indexing 4.286625836 seconds

Figure 6.3 Without Indexing

6.4 Graph Comparisons

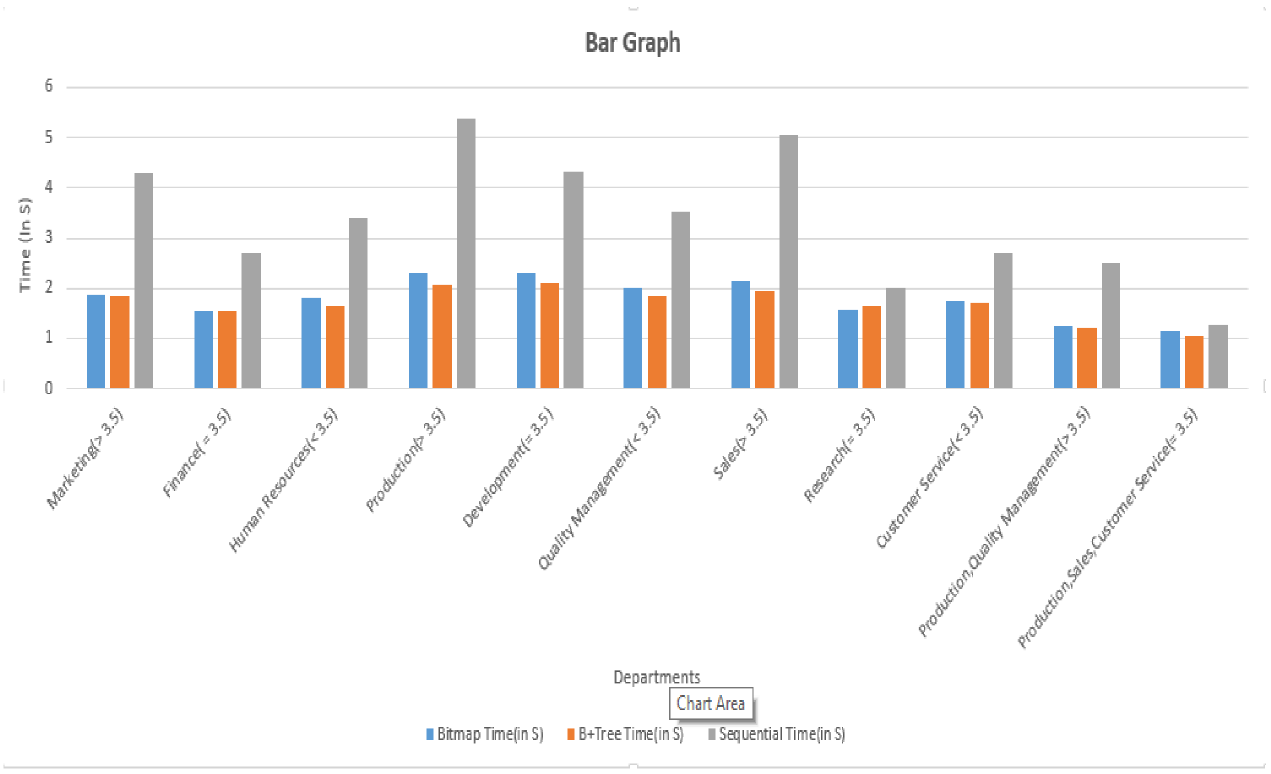


Figure 6.4 Bar Graph

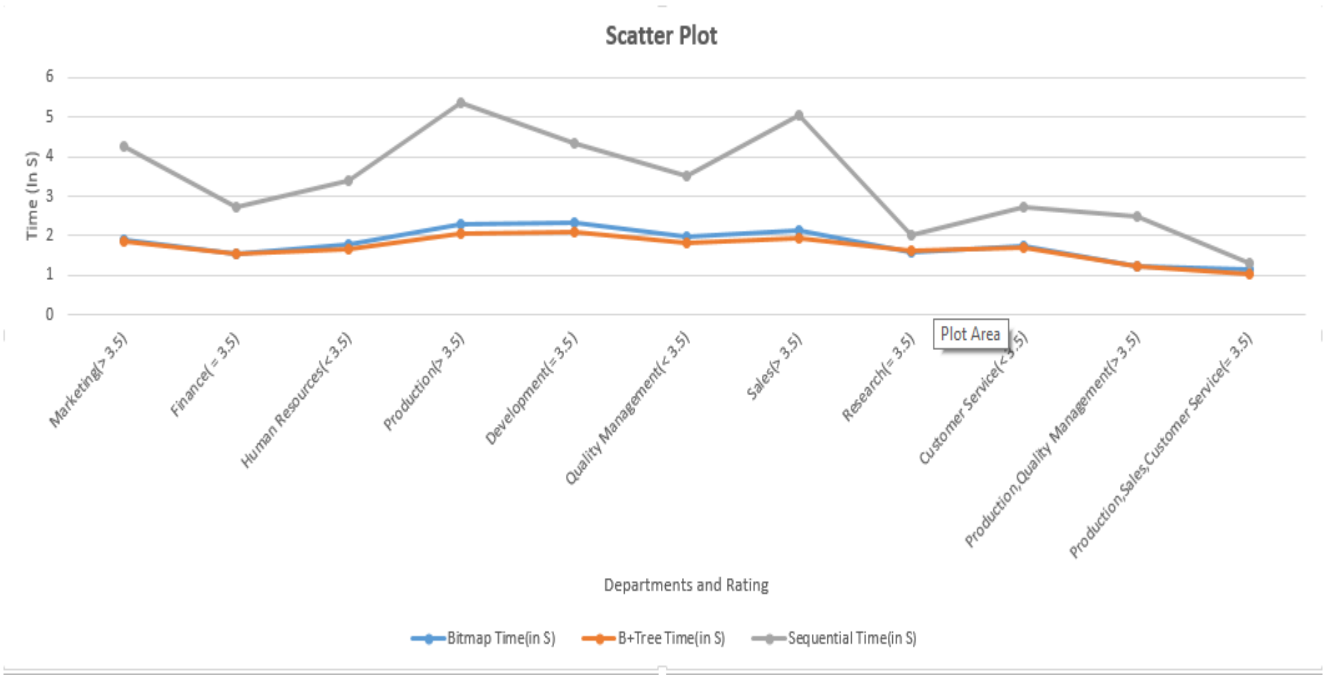


Figure 6.5 Scatter Plot

CHAPTER 7: CONCLUSIONS AND FUTURE WORK

It is commonly accepted that the Bitmap index is more efficient for low cardinality attributes. Our experiment shows that the Bitmap index effectively reduces the query response time for a column with high cardinality compared B^+Tree index. We have also shown that Bitmap index file size and index creation time grow gradually as the column cardinality increases as compared to B^+Tree which grows significantly. Also, we have demonstrated that although the index file size of the bitmap index is affected by column cardinality; the query processing time is constant as the column cardinality increases. Besides, the Bitmap index is also efficient for other types of queries, such as joins on keys, multidimensional range queries, and computations of aggregates. Thus, we conclude that the Bitmap index is the conclusive choice for a DW designing no matter for columns with high or low cardinality.

It is often considered that I/O cost dominates the query response time. Moreover, the main memory size may play a role in index performance as a small memory size might trigger a lot of paging activities, which then could change the query performance of both indexing. Thus, our future work includes the evaluation of I/O costs on an upgraded hardware system.

REFERENCES

[1] Chan, C.-Y., & Ioannidis. "Bitmap Index Design and Evaluation." Volume 27 Issue 2, June 1998 New York: ACM Proc. SIGMOD Conf. 1998. pp. 355-366

[2] Chan, C.-Y., & Ioannidis, "An Efficient Bitmap Encoding Scheme for Selection Queries", New York: ACM Proc. SIGMOD Conf. 1999. pp. 215-226

[3] Johnson, T. "Performance Measurements of Compressed Bitmap Indices". Jan 2002 Massachusetts: Kluwer Academic Publishers Norwell Proc. Int'l Conf. On Very Large Data Bases (VLDB) .1999. pp. 278-289

[4] Stockinger, K., & Wu, K., & Shoshani. "Evaluation Strategies for Bitmap Indices with Binning" August 30-September 3, 2004 Spain: Proc. Int'l Conf. Database and Expert Systems Applications (DEXA). 2004. pp. 120-129

[5] Wu, K., & Otoo, E.J., & Shoshani, A. "Compressing Bitmap Indexes for Faster Search Operations", July 2002. Washington, DC, USA Proc. Int'l Conf. Scientific and Statistical Database Management (SSDBM). 2002. pp. 99-108

[6] Wu, M.-C., & Buchmann, A.P. "Encoded Bitmap Indexing for Data Warehouses", June 1998 New York: ACM Proc. Int'l Conf. Data Engineering (ICDE). 1998. pp. 220-230

[7] P. O'Neil and G. Graefe, A. "Multi-Table join through Bitmapped join indices", Volume 24 Issue 3, Sept. 1995 New York: ACM Proc. SIGMOD Conf. 1995. pp. 8-11

[8] Raghurama Krishnan, J. Gehrke, Database Management Systems, TATA McGraw-Hill 3rd Edition, 2002. pp. 485-500

[9] D.A. Patterson and J.L. Hennessy, Computer Organization and Design: The hardware-software interface, 3rd Edition, 2005. pp. 574-579

[10] William Stallings, Operating Systems – Internal and Design Principles, Pearson education/PHI, 5th Edition, 2005. pp. 558-562

[11] Alex Berson and S.J. Smith, Data Warehousing, Data Mining and OLAP, TATA MGH, 3rd Edition, 2004. pp. 63-66

[12] V. Sharma, Bitmap index vs. b-tree index: Which and when, http://www.oracle.com, 2006.

[13] S. Chaudhuri, U.Dayal. "An Overview of Data Warehousing and OLAP Technology", Volume 26 Issue 1, March 1997 New York: ACM SIGMOD RECORD. 1997. pp. 65-74