

PERFORMANCE STUDY OF B, B* & B⁺ TREES IN DATABASE INDEXING

by

Maria Wahid Chowdhury

Student Number: 9505028F, Session: 1994-95-96

For the partial fulfillment of M. Sc. Engineering Degree in
Computer Science and Engineering



Supervised by

Dr. Chowdhury Mofizur Rahman

Head and Associate Professor, CSE Department
BUET



#94603#

Department of Computer Science and Engineering (CSE)
Bangladesh University of Engineering and Technology (BUET)
Dhaka-1000, Bangladesh

Performance Study of B, B*, B⁺ trees in Database Indexing

A Thesis Submitted by

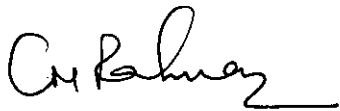
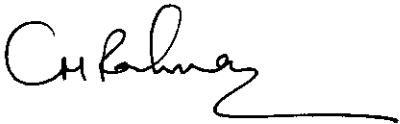



Maria Wahid Chowdhury

Roll No. 9505028F, Session: 1994-95-96

For the partial fulfillment of the degree of M. Sc. Engineering in
Computer Science and Engineering

Examination held on
November 16, 2000.

Approved as to style and contents by:

- 
1. (Dr. Chowdhury Mofizur Rahman) (Supervisor) Chairman
Associate Professor
Department of CSE, BUET, Dhaka
- 
2. Head Member
Department of CSE, BUET, Dhaka
- 
3. Dr. Md. Abul Kashem Mia Member
Assistant Professor
Department of CSE, BUET, Dhaka
- 
4. Dr. Md. Saidur Rahman Member
Assistant Professor
Department of CSE, BUET, Dhaka
- 
5. Dr. Anisul Haque Member (External)
Associate Professor
Department of EEE, BUET, Dhaka

Acknowledgement

I like to express my sincerest appreciation and profound gratitude to my supervisor Dr. Chowdhury Mofizur Rahman, Head & Associate Professor, Department of Computer Science and Engineering, BUET, for his supervision, encouragement and guidance. His keen interest in this topic and valuable suggestions and advice were the source of all inspirations to me. I also like to convey gratitude to all my course teachers here. Their teaching helps me a lot to start and complete this thesis work. Finally, I would like to acknowledge the assistance and contribution of a large number of individuals and express my gratefulness to them.

Maria Wahid Chowdhury

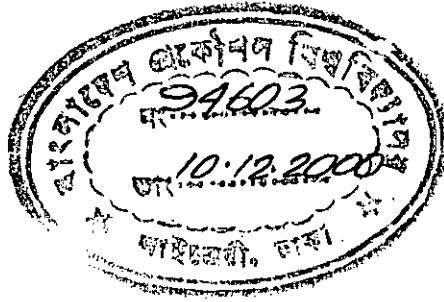
Abstract

Complex accessing structures like indexes are a major aspect of centralized databases. The support for these structures is the most important part of the database management systems (DBMS). The reason for providing indexes is to obtain fast and efficient access to data. Most of the centralized database management systems use B-tree or other types of index structures. But in distributed databases no index structure is used to obtain fast and efficient access to the data. Therefore efficient access is the major problem in distributed databases. We proposed a distributed index model, which is a data structure based index comprising of two types of index structures: Global Index (GI) and Local Index (LI). GI is created and maintained by distributed database component (DDB) and LI is created and maintained by local database component (DB) of a distributed database management system. B-tree was used to implement both GI and LI. A simulation program tested the proposed model and found satisfactory results.

TABLE OF CONTENTS

1	INTRODUCTION	3
1.1	DATABASE AND DBMS	3
1.2	INDEXING	4
1.3	B, B [*] , AND B ⁺ TREES	4
1.4	DISTRIBUTED DATABASE	6
1.5	FINDING RECORDS IN A DISTRIBUTED DATABASE	6
1.6	PROPOSED DISTRIBUTED INDEX	7
2	DATABASE INDEXING	8
2.1	INTRODUCTION	8
2.2	STATIC INDEXES	11
2.2.1	<i>Organization of the index</i>	11
2.2.2	<i>Insertions</i>	13
2.2.3	<i>Physical Organization of Overflow Area</i>	16
2.3	DYNAMIC INDEXES	16
2.3.1	<i>Binary Trees</i>	17
2.3.2	<i>AVL Trees</i>	17
2.3.3	<i>Multivay Trees</i>	18
2.3.4	<i>B-trees</i>	18
2.3.5	<i>B[*]-trees</i>	38
2.3.6	<i>B⁺-trees</i>	43
2.4	COMPARISON OF STATIC AND DYNAMIC INDEXES :	48
3	EXPERIMENTAL RESULT ON PERFORMANCE STUDY OF B, B[*], AND B⁺ TREES	50
3.1	INTRODUCTION	50
3.2	SIMULATION PROGRAM	50
3.3	RESULTS	51
3.4	CONCLUSION	58
4	DISTRIBUTED DATABASE	59
4.1	INTRODUCTION	59
4.2	DISTRIBUTED DATABASE	59
4.3	DISTRIBUTED DATABASE MANAGEMENT SYSTEMS (DDBMS)	61
4.4	REFERENCE ARCHITECTURE FOR DISTRIBUTED DATABASES	62
4.5	TYPES OF DATA FRAGMENTATION	65
4.5.1	<i>Horizontal Fragmentation</i>	65
4.5.2	<i>Vertical Fragmentation</i>	66
4.6	FINDING RECORDS IN A DISTRIBUTED DATABASE	67
5	PROPOSED DISTRIBUTED INDEX	69
5.1	INTRODUCTION	69
5.2	DISTRIBUTED INDEX ARCHITECTURE	69
5.3	DISTRIBUTED INDEX SEARCHING (GI SEARCHING)	72
5.4	GI INSERTION	74
5.5	GI DELETION	76
6	EXPERIMENTAL RESULT OF PROPOSED DISTRIBUTED INDEX	78
6.1	INTRODUCTION	78
6.2	SIMULATION PROGRAM	78
6.3	RESULTS	78
6.3.1	<i>Searching account number 333</i>	87

6.3.2	<i>Searching account number 489</i>	87
6.3.3	<i>Searching account number 000</i>	88
6.3.4	<i>Deleting account number 490</i>	88
6.3.5	<i>Deleting account number 380</i>	89
6.3.6	<i>Inserting the account number 697 at site 7</i>	90
6.3.7	<i>GI after site 10 has been vanished</i>	91
6.3.8	<i>GI after site 8 has been vanished</i>	91
7	CONCLUSION	93
8	REFERENCES	94



1.1 Database and DBMS

Database is a collection of related data in a specific structure of any organization or corporation. Underlying structure of a database is the concept of a data model, a collection of tools for describing data, data relationships, data semantics, and consistency constraints. The various data models, such as object-based logical models, record –based logical models, and physical data models have been used.

Relational data model is a record-based data model and is used in describing data at the conceptual and view levels. In this model database structured in a fixed format records of several types. Each record type defines a fixed number of attributes and each attribute is usually of fixed length. The use of fixed length record simplifies the physical-level implementation of the database. This makes the relational data model most popular data model. A relational database consists of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values [17].

Although databases come in a variety of sizes, almost any collection of data can become quite unwieldy as it receives and stores more and more data over time. Computers make it possible to harness large data collections efficiently using a Database Management System (DBMS). A DBMS is a program, or collection of programs, that allows any number of users to access data, modify it (if necessary), and construct simple and complex requests to obtain and work with selected records. Data management tasks of DBMS fall into one of three general categories:

- Entering data into the database
- Reordering the records in the database
- Obtaining subsets of the data

Actually, there are lots of tasks anybody can perform with a DBMS, including creating and designing the database itself. A DBMS's biggest asset is its ability to provide ex-

tremely quick access and retrieval from large database. To determine the positions of the specific records in a database, DBMS use a special technique called index.

1.2 Indexing

An index for a data file in the database system works in much the same way as a catalog for a book in a library. If we are looking for a book by a particular author, we look in the author catalog, and a card in the catalog tells us where to find the book. To assist us in searching the catalog, the library keeps cards in alphabetic order, so we do not have to check every to find the one we want. To gain fast access to the records in a data file, we can use an index structure. Each index structure is associated with a particular search key (one or more attributes of the data). Just like a library catalog, an index stores the values of the search keys in sorted order, and associates with each search key the records in the data file that contain that search key. A data file may have several indices, on different search keys.

Indexing is a data structure based technique for accessing records in a file. It is a data structure technique in the sense that a search of a data structure yields the required address. In this technique a main file of records is supplemented by one or more indexes. Indexes may be the part of the main file or be separate files, and may be created and destroyed as required without affecting the main file. When changes are made to the main file, appropriate update operations must be carried out on any indexes to that file. There are two main types of indexes such as static index and dynamic index. We elaborately discussed both of these types and their various ways of implementations in the Chapter 2.

1.3 B, B*, and B⁺ Trees

Instead of binary trees multiway trees can be used for implementing better dynamic indexes. Multiway Trees are a generalization of binary trees. Instead of containing a record and two pointers, as in a binary tree, a node contains R records and R+1 pointers. This alleviates the long retrieval times found with binary trees. The increase in the branching factor typically makes the tree shorter than the corresponding binary tree for the same number of records. However, complex balancing operations may be required as records are inserted and deleted [18].

B-trees were devised by Bayer and McCreight [5]. They have neither the retrieval nor the maintenance problems of binary trees because they are multiway trees with efficient self-balancing operations.

B-trees are balanced multiway trees. A node of the tree may contain several records and pointers to "children". We use term "child" to refer to the immediate descendent of a node; hence "siblings" refers to nodes with the same parent. The operations of retrieval, insertion, and deletion are guaranteed efficient even in the worst case.

A number of variations on the B-tree data structure have been devised. Typically, each is designed to overcome some of the deficiencies of the B-tree. We considered two such variations: B^* -trees, which arise from a suggestion by Bayer and McCreight [5], and B^+ -trees, which were suggested by Knuth [13].

The B^* -tree performs searches in the same way as the B-tree, except that there are different operation of a B-tree more efficient by reducing the number of occasions when a node had to be split. If the node into which we need to insert a record is full, we might in certain circumstances be able to solve the overflow problem by local redistribution of records rather than by splitting the nodes.

Knuth proposed a variation on B trees that for clarity Comer designed the B^+ -tree. Records in a B^+ -tree are held only in the terminal nodes of the tree. The terminal nodes are linked together to facilitate sequential processing of the records and termed the sequence set. Non-terminal nodes are indexes to lower levels in a similar way to that of B trees. Nodes in the index levels contain only key values and tree pointers. There is no need for the terminal nodes' tree pointer fields. Thus terminal nodes have a different structure from non-terminal nodes. We discussed B, B^* , and B^+ trees and their numerous algorithms in the Chapter 2.

We did a simulation program for constructing and using B, B^* , and B^+ trees for database indexing based on the available structures and algorithms. This program generates random data to insert, search, and delete. Performance of each tree is measured in terms of number of comparisons, number of nodes, number of splits and the height of the tree re-

quired. The results of the simulation program were recorded and plotted separately. We showed these results in the Chapter 3.

1.4 Distributed Database

In recent years, distributed databases have become an important area of information processing, and it is easy to foresee that their importance will rapidly grow. There are both organizational and technological reasons for this trend: distributed databases eliminate many of the shortcomings of centralized databases and fit more naturally in the decentralized structures of many organizations. A distributed database is a collection of data, which belong logically to the same system but are spread over the sites of a computer network. Each site of a network has autonomous processing capability and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem. The global schema defines all the data, which is contained in the distributed database as if the database were not distributed at all. Each global relation can be split into several non-overlapping portions, which are called fragments. Fragments are logical portions of global relation, which are physically located at one or several sites of the network [7]. We discussed distributed database in details in the Chapter 4.

1.5 Finding records in a Distributed Database

At present distributed databases are inefficient in locating records since it is not using any global index structure. Distributed Database Management Systems collect the pertaining fragments from different sites and reconstruct the data file as if it were not fragmented. After reconstruction every record is read and checked one by one to identify the desired record. Alternatively, Distributed Database Management Systems can search the record in every fragments one after another. In both of these cases lot of extra works need to be done to find a record. To eliminate these extra works during searching a record an efficient index structure can be used.

1.6 Proposed Distributed Index

Most of the centralized database management systems use B-tree or other types of index structures. But in distributed databases no index structure is used to obtain fast and efficient access to the data. Because it is very difficult to build and maintain such structures and because it is not convenient to navigate at a record level in distributed databases. Therefore, efficient access is the main problem in distributed databases. To provide efficient access to the data we proposed a distributed index concept. Distributed index is also a data structure based index comprising of two types of index structures. One is Global Index (GI) and the other is Local Index (LI).

GI is created and maintained by distributed database component (DDB) of distributed database management systems (DDBMS). LI is created and maintained by local database management component (DB) of DDBMS. We preferred B-tree for implementing both GI and LI of distributed index. For every site there is a Local Index (LI), which has been created, updated and used independently. Like other local database management components LI enjoys autonomy in each site. There must be a single global index (GI) for a distributed index. GI is created, updated and used based on local indexes. All the local indexes are perfectly mapped with the global indexes. When a record is searched in a distributed database, GI used first to determine which LI needs to be used to find the data. After selecting the right LI it is used to access records in the corresponding site. The records in a GI node are different from that of a LI. LI holds the index value and the pointer/address of the concern record in the data file. But GI record holds the minimum and maximum index values of a local index and the pointer/address of that local index. We explained the proposed distributed index structures and its algorithms in Chapter 5.

We developed the second simulation program for constructing and using the proposed distributed index. We showed the results of the program in Chapter 6.

We proposed and examined the structure and algorithms for distributed index considering simple distributed databases. Complex distributed databases will certainly require extra efforts for indexing.

DATABASE INDEXING

2.1 Introduction

Many queries reference only a small proportion of the records in a data file. For example, the query "Find all accounts at the Perryridge branch" references only a fraction of the account records in the Account data file. It is inefficient for the database systems to have to read every record and to check the branch name field for the name "Perryridge". We need database systems should be able to locate these records directly and for this we need to keep additional structure i. e. index structure, associated with the data file.

Indexing is a data structure based technique for accessing records in a file. It is a data structure technique in the sense that a search of a data structure yields the required address. In this technique a main file of records is supplemented by one or more indexes. Indexes may be the part of the main file or be separate files, and may be created and destroyed as required without affecting the main file. When changes are made to the main file, appropriate update operations must be carried out on any indexes to that file.

Index files can be compared with the index or table of contents of a book. Consider the index of a book. It consists of a number of entries, each of which is a pair:

Topic, page number(s)

Book indexes are usually arranged alphabetically, which makes it easy to find a particular topic and hence the pages on which it are mentioned. An index file is like a book index. Index files typically contain records of the following form:

Key value, pointer(s) to the main file

The pointers reference records in the main file that have the particular key value. Here we will discuss using data structure techniques rather than computational techniques to solve the file accessing problem. We consider the efficiency of the data structures and the operations required to maintain optimum efficiency. It is important to remember that when we discuss "records" in an index, for example in a node of a search tree, we mean records of the form shown above rather than records of the main file.

Although indexes to sequential files are common, non-sequential files can also be indexed. A sequential file is typically in sequence only with respect to one key. For example, a file of insurance records ordered by policy number would be non-sequential with respect to a second key such as name of policyholder.

Without data structures, processing records with respect to a key other than the one by which the file is sequenced is likely to be inefficient. Consider the file of insurance records. To find a record when given the name of the policyholder, we would probably have to perform a linear search through the file. If we need a list of policyholders according to date of birth, we would probably have to perform a sort operation.

Except in certain books, topics in the body of a book do not normally appear in alphabetical order. Books can therefore be regarded as non-sequential from the point of view of topic. Indexes help to locate topics. In the remainder of this chapter we consider index organization in the context of indexed sequential files.

A sequential file is one in which records can be accessed in sequential order, which is usually primary key order. An indexed sequential file is sequential file supplemented by an index structure. The purpose of the index is to speed up access to a particular record. Normally the index is effective only when the file is being indexed is stored on a direct-access device. Un-indexed sequential files on the other hand, could reasonably be stored on serial devices.

There are alternatives to indexing as a technique for achieving fast access to sequential files, but they tend to be comparatively slow or restrictive. A binary search is one possibility, but it requires that for a given I it must be possible to compute the address of the I th record in the file. If sequencing is implemented by pointers rather than physical adjacency, this may not be feasible. Even if it is possible to compute the address, performance of a binary search algorithm is not impressive. Suppose that there are M logical records in a file and that they are packed N to a physical record. The average number of physical accesses required to find a record using a binary search is about $(\log_2 \lceil M/N \rceil) - 1$.

The way we organize an index file will depend on the operations we wish to perform via key field. Possible operations include retrieval of individual records and processing of all records in the key order. If the index is used only to locate a record with a particular value of a key, then hashing may be suitable way of organizing it. However, if the index is the means of accomplishing both sequential processing and fast individual record access, then a tree structure is a better choice. This is because sequential processing is accomplished by simple tree traversal and the nature of the tree also allows a particular record to be located quickly. The indexes we examine here are based on trees.

Because the main file is sequenced, it may not be necessary for the index to have an entry for each record. Figure 2.1 shows a sequential file with a two level index. Level 1 of the index holds an entry for each three-record section of the main file. In a similar way, level 2 indexes level 1. It may be sufficient for an index to a sequential file simply to identify the part of the main file containing the desired record. The final part of a search operation can then be a simple linear or binary search of the identified section. Typically this will be a search in main memory, of a physical record retrieved from the file. Consider the book analogy. The table of contents acts as an index in our sense of term. The file (book) is in sequence by the key being indexed (chapter and section numbers). Section numbers in the body of the text may be long (for example, p.q.r.s.t). In the table of contents, however, entries may be limited to p.q.r, leaving the reader to interpolate to find more precise sub headings. Thus while there is still a search space, it has been reduced through the use of the index.

Indexed sequential organization is straightforward apart from the problem of inserting new records into the file. The problems are to preserve the sequence of the records and update the index appropriately. Two broad classes of solutions to these problems are static index techniques and dynamic index techniques.

Although the contents of a static index may change as we perform insertions and deletions on the main file, the structure of the index does not change. Typically the insertion algorithm uses overflow areas. This method however, tends to lead to gradual loss of efficiency for search and update operations, and periodic maintenance is required to restore performance. Typically this involves running a standalone program that writes a new ver-

sion of the main file eliminating overflow lists. The index is rebuilt during the rewriting. The IBM ISAM system uses static index structures [18].

A **dynamic index** adapts as records are inserted into and deleted from the main file. In some sense, maintenance of efficiency is an integral part of the insert and delete algorithms. There is no need to run a separate maintenance program periodically. Dynamic indexing methods are characterized by the splitting and joining of nodes in the index tree. The IBM VSAM system uses dynamic indexing.

Both static and dynamic indexes are useful depending on the type of application. We will compare their efficiency out the following operations:

1. Searching for a record with a given key
2. Inserting a new record
3. Deleting a record with a given key.

2.2 Static Indexes

One approach to organize a tree in a static index structures is to keep its structure fixed and to deal with insertions by means of overflow lists at the leaves.

2.2.1 Organization of the index

A static index can be regarded as a series of fixed size. The lowest-level table indexes the main file itself, the next highest level indexes the lowest level, and so on. Figure 2.1 shows a small example file with two levels of indexing. A level 1 index entry holds the highest key value in a three-record section of the main file together with a pointer to the section. A typical entry is

42, <pointer to the section with 42 as its highest key>

The choice of three for the size of the main file section is arbitrary here. In practice it is likely to be related to the size of physical and logical records. For example, if the storage device can transfer up to 1024 bytes in one access and logical records are 80 bytes long, each main file section will probably contain 12 records.

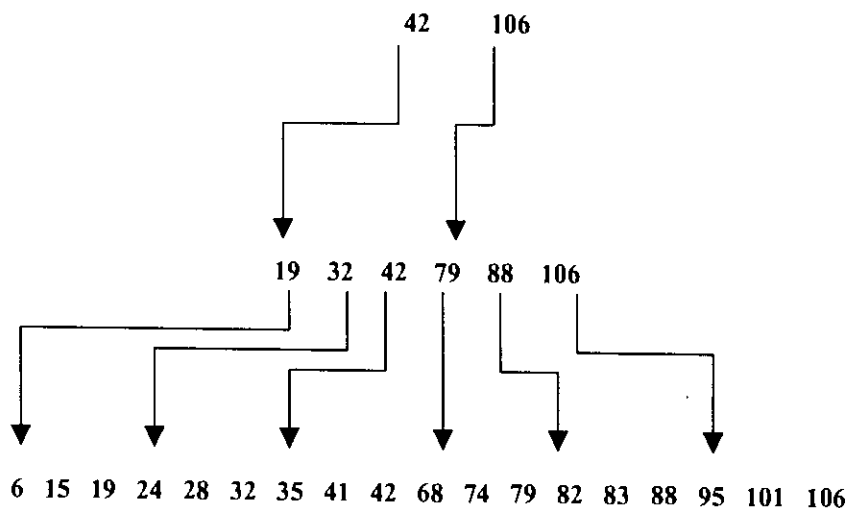


Figure 2.1 File with two level index

Thus a complete section can be read in one disc access. The index holds the highest key value in each three record section of the level 1 index. Again, the choice of three here for section size is arbitrary. Given particular file characteristics and properties of the storage media, it is possible to calculate the number of index levels and the total space they occupy.

To see why do we index a section by it's highest rather than its lowest key; let us follow the retrieval of the record with key 28 from Figure 2.1. We begin at the top of the tree with the level 2 index. We select the smallest index entry with key greater than or equal to the target key. In this select 42. The associated pointer points to the level 1 section that contains the following entries:

- 19→
- 32→
- 42→

Using the same selection criteria, we follow the pointer associated with the entry 32 in the level 1 index. The pointer gives us the address of the section in the main file containing

- 24
- 28
- 32

When we search this, we find the record with key 28. If we had been looking for a record with key 29 instead, the search would fail when key 32 was encountered.

By holding the highest rather than lowest key in the index, we avoid an extra comparison at each index level during a search operation. This is because the pointer to the lowest value and the key of the largest value together defines a search subspace. For example, the pointer associated with entry 32 in the level 1 index points to the section starting with 24. Any record with a key in the range of 24 through 32 will be in that section. If we hold the lowest key, we would have to look at the next index entry to establish the upper bound on the subspace.

2.2.2 Insertions

Next, let us see the way in which the file and indexes change when insertions are made into the main file. We will insert records with keys 7, 33, and 18. These insertions will cause overflow conditions that must be resolved.

Insert 7.

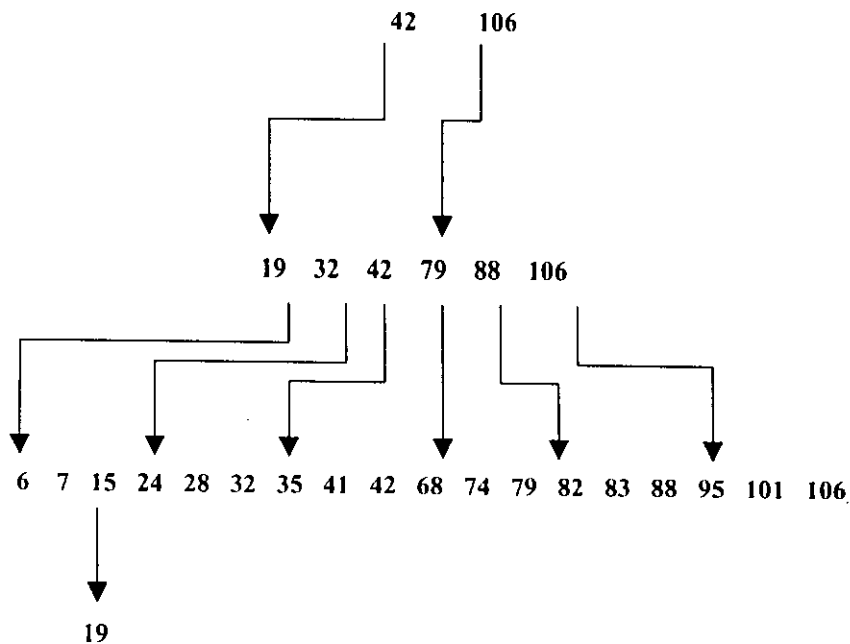
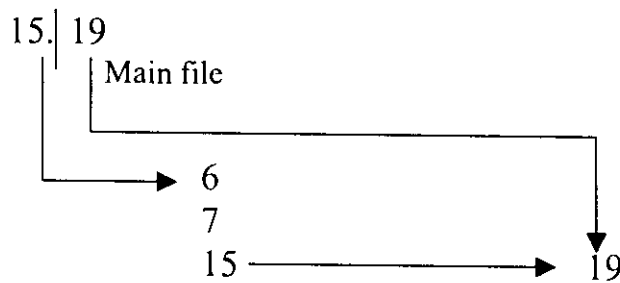


Figure 2.2 File with overflow

When we insert a record with key 7, the index structure leads us to the first section of three records in the main file. This is where the record with key 7 would be if it were in the file. To preserve the record sequence, the new record must be inserted after the one with key 6. Therefore, the record with key 19 is moved out of the main file section and into an overflow area. Figure 2.2 shows the new configuration.

We assume that there is space at the end of each section of records in the main file or a pointer to a list of records that have been moved out of the section. Observe that sequential processing of the file slowed by the need to make an access to the overflow area. Between records with keys 15 and 24. However, access to an arbitrary record need not be slowed if we modify the level 1 index. Suppose that in addition to holding the highest key in either the main file section or its overflow list, it also holds the highest key in the main file section alone. With this information we can tell whether to look for a particular record in the main file or in the overflow area. In the case of our example the level 1 index entry for the first main file section is now

Index



The entry indicates that 15 is the highest key in the section and that 19 is the highest key when the overflow list is taken into account. To go directly from the level 1 index entry to the list. However, in subsequent diagrams we will show the key values in the level 1 index entries but omit pointers to overflow lists to prevent the diagrams from becoming cluttered.

Insert 33.

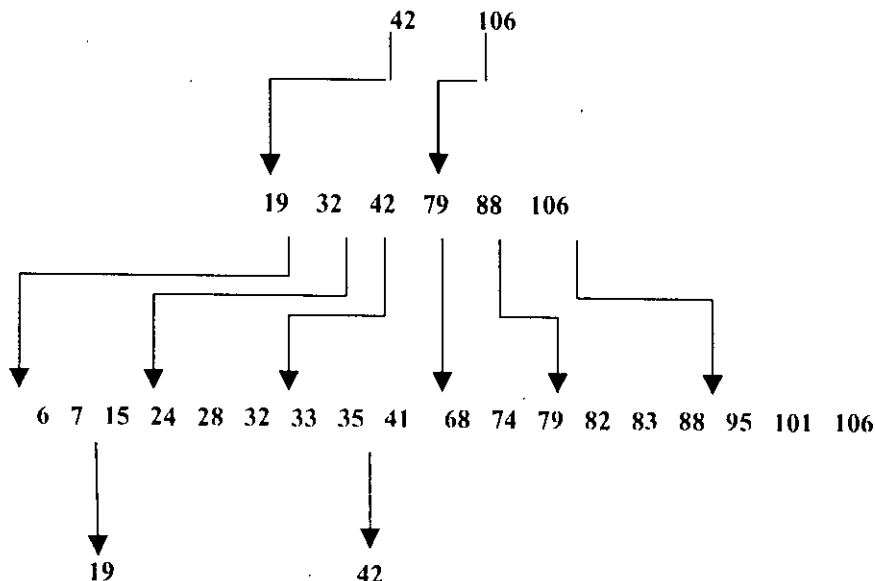


Figure 2.3 File with overflow

Key 33 lies between the ranges of the records stored in the second and third three-record sections of the main file. Examination of the level 1 index indicates that if the record with key 33 were already in the file it would be in the third section; its key is greater than the largest key recorded for the second section. Hence the new record is inserted into the third section; Figure 2.3 shows the new configuration. Although the two overflow lists are separate in the diagrams, there is no reason why they should not be stored interleaved in the same file.

Insert 18.

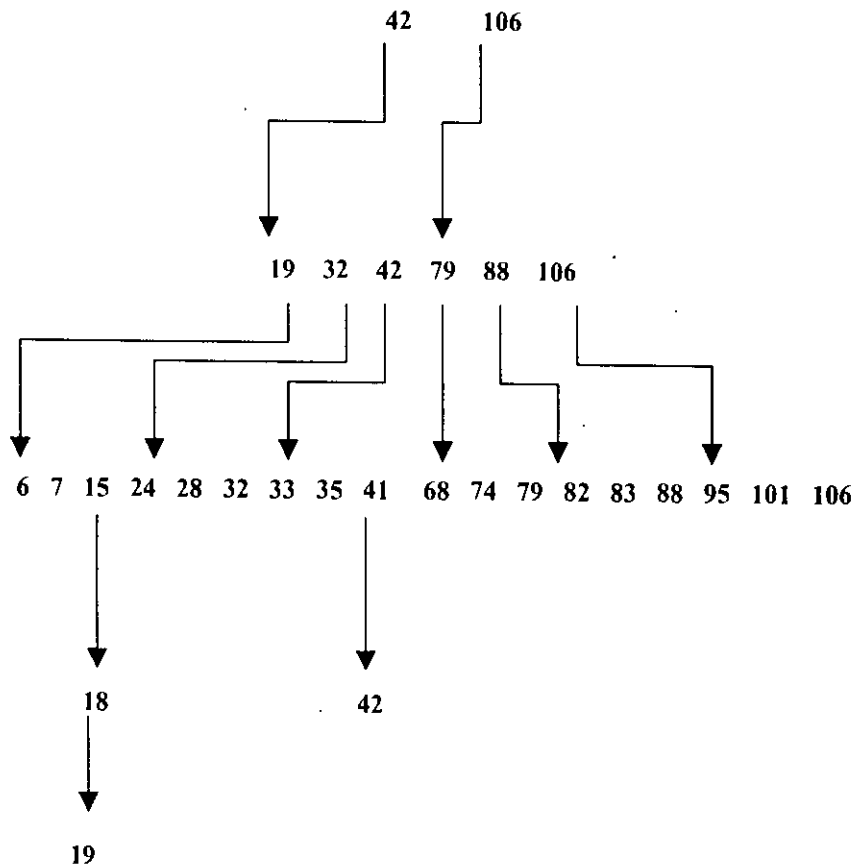


Figure 2.4 Overflow chain

Key 18 is lower than the highest key associated with the first file section (19) but higher than the last key in the main file section (15). The new record is therefore put directly into the overflow list. Figure 2.4 shows the new configuration. Although sequential accessing of the file may not be slowed further as a result of this insertion, a retrieval of record with key 19 is likely to take longer than before.

2.2.3 Physical Organization of Overflow Area

Above, we discussed in abstract terms how insertions into the main file might be handled. How might an overflow scheme be implemented? One possibility is to write the initial sequential file cylinder by cylinder on a disc, leaving a number of tracks free on each cylinder. Records overflowing from sectors in a cylinder would be placed in the spare tracks in the same cylinder, that is, in a cylinder overflow area. In this way no disc head movement would be required when following a pointer from main file. If overflows are confined to such areas, sequential processing is reasonably fast. However, what happens if there are enough insertions in one part of the file to exhaust a particular cylinder overflow area? Such clustered insertions with a small range of key values compared with the key range of the file as a whole are a problem for indexed sequential file organizations. In a static organization, such as described above, they will result in long overflow lists. One solution is to provide a number of spare cylinders that can be used by overflows from any cylinder. However, if these have to be used, both sequential and direct processing of the file begins to require time-consuming disc head movements. Performance degrades rapidly.

A weakness of the static organization is its potential degradation of performance as insertions are made. In a typical application we would expect that there would be more retrievals than insertions or deletions, so it is important that they be as efficient as possible. In a static organization maintenance programs may have to be run periodically to restore performance levels. During these runs the file is likely to be inaccessible. In contrast, dynamic indexes, which we discuss next, may gradually change shape in order to preserve efficiency. Compared with static indexes, more work may be done when a record is inserted or deleted, but there is no need for separate periodic maintenance [18].

2.3 Dynamic Indexes

Many dynamic indexes are implemented as trees. We consider four common tree structures and compare them in terms:

- Depth (minimum for given number of records)
- Ease of maintenance
- Maximum order

The four tree types are binary, AVL, multi-way, and B-trees. We adhere to the convention of depicting a tree with the root uppermost and regard the root as the top of the tree.

2.3.1 Binary Trees

Binary trees suffer from two disadvantages compared with other trees: long retrieval times and effort needed to maintain efficient access. Binary trees have a branching factor of two, that is, each node has at most two immediate descendants (children). Consequently, the minimum height of a tree containing N records is $\lfloor \log_2 N \rfloor + 1$. For example, a tree of 100 records has at least seven levels. If a tree is held on secondary storage, then there tends to be a proportional relationship between the numbers of node reads and the number of physical seeks. It is therefore desirable to have short trees to minimize the number of physical accesses. Because of the small branching factor, binary trees tend to be tall. For best performance the tree should be balanced in the sense that the sum of the lengths of the paths from the root to the nodes is minimized. After an insertion or a deletion the tree may have to be rebalanced. The operations required to balance an arbitrary tree are relatively complex.

2.3.2 AVL Trees

AVL trees, devised by Adel'son- Vel'skii and Landis [3], are restricted growth binary trees. They were invented as a solution to the balancing problem encountered with normal binary trees. An AVL tree is not necessarily perfectly balanced. In a perfectly balanced binary tree the number of nodes in the two sub trees of an arbitrary node differ by at most 1. The balancing operations are simpler than those for ordinary binary tree, but AVL trees still have comparatively long search times. Bounds have been established for the height of an AVL tree containing N records as follows:

$$\log_2(N+1) \leq \text{height} \leq 1.4404 \log_2(N+2) - 0.328$$

Considering again a tree with 100 records, we have

$$6.658211 \leq \text{height} \leq 9.282961$$

The search problem persists because we still have tall trees.

2.3.3 Multiway Trees

Multiway Trees are a generalization of binary trees. Instead of containing a record and two pointers, as in a binary tree, a node contains R records and $R+1$ pointers. This alleviates the long retrieval times found with binary trees. The increase in the branching factor typically makes the tree shorter than the corresponding binary tree for the same number of records. However, complex balancing operations may be required as records are inserted and deleted.

2.3.4 B-trees

B-trees were devised by Bayer and McCreight [5]. They have neither the retrieval nor the maintenance problems of binary trees because they are multiway trees with efficient self-balancing operations.

B-trees are balanced multiway trees. A node of the tree may contain several records and pointers to "children". We use term "child" to refer to the immediate descendent of a node; hence "siblings" refers to nodes with the same parent. The operations of retrieval, insertion, and deletion are guaranteed efficient even in the worst case. B-tree definition

We follow Knuth [13] rather than Bayer and McCreight and define a B-tree of order M to be a tree with the following properties:

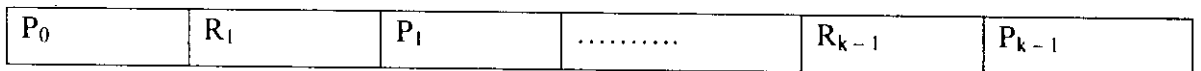
1. No node has more than M children.
2. Every node, except for the root and terminal nodes, has at least $\lceil M/2 \rceil$ children.
3. The root, unless the tree only has one node, has at least two children.
4. All terminal nodes appear on the same level, that is, they are at the same distance from the root.
5. A non-terminal node with k children contains $k - 1$ records. A terminal node contains at least $\lceil M/2 \rceil - 1$ records and at most $M - 1$ records.

We are considering index structures: a record in the tree will therefore consist of a key and a pointer to the main file. We can speak of a B-tree of order 8 ($M=8$), a B-tree of 197 ($M=197$), and so on. The integer M imposes bounds on the "bushiness" of the tree. While the root and terminal nodes are special cases, normal nodes have between $\lceil M/2 \rceil$ and M children and $\lceil M/2 \rceil - 1$ and $M - 1$ records. For example, a normal node in a tree of order

11 has at least 6 and not more than 11 children. The lower bound on the node size ensures that the tree does not get too tall and thin, since this results in slow searches. The upper bound on the node size ensures that the searches of an individual node will be fast. When implementing the tree, for example, as a file of records, the upper bound allows us to define an appropriate record type. The lower bound ensures that each node is at least half full and therefore that file space is used efficiently.

The definition above only determines the structure of a B-tree; to be useful there must be some ordering of the records in the tree. In what follows we will make the following assumptions:

1. Within a node of $K - 1$ records, records are numbered $R_1, R_2, R_3, \dots, R_{k-1}$ and pointers to children are numbered $P_0, P_1, P_2, \dots, P_{k-1}$. Thus a typical node may be depicted as follows:



2. Records in the sub tree rooted in P_0 have keys less than the key of record R_1 . Records in the sub tree rooted in P_{k-1} have keys greater than the key of record R_{k-1} . Records in the sub tree rooted in $P_{i(0 < i < k-1)}$ have keys greater than the key of record R_i .

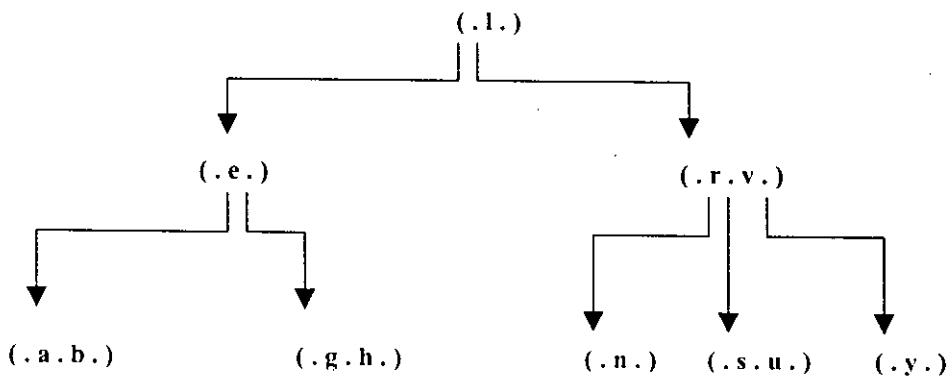


Figure 2.5 Example of B-tree

Figure 2.5 shows an example B-tree. In this and the subsequent examples we will assume that the keys of the records in the tree are single characters.

It is not always possible to determine the order of a tree by looking at it. The tree of Figure 2.5 must be at least order 3 because some non-root nodes have three children. At the

same time it must be less than order 5 because some nodes have only two children. It is therefore a tree of order 3 or order 4.

2.3.4.1 B-tree terminology

We will term **adjacent siblings** two nodes that have the same parent and are pointed to by adjacent pointers in the parent. Thus in the tree of Figure 2.5 (.n.) and (.s.u.) are adjacent siblings, whereas (.n.) and (.y.) are not adjacent siblings. Adjacent siblings are pointed to by P_{i-1} and P_i (for some i). We will term record R_i the separating record for two siblings. Thus in the tree of Figure 2.5 adjacent siblings (.s.u.) and (.y.) are separated by record with key v .

2.3.4.2 Searching B-trees

When searching for a record with a given key, we start by examining the root node. We search the node for the required record. If the record is not found, comparisons with the keys in the node will identify the pointer to the sub tree that may contain the record. If the selected pointer is null, then we are at the lowest level in the tree and the record we are searching for is not present in the tree. If the pointer is not null, then we read the node pointed to, that is, the root node of the sub tree, and repeat the operation. Algorithm 2.1 contains a pseudo-code algorithm for the search.

2.3.4.3 Performance of the B-tree search algorithm

Assume the B-tree is of order M and that it contains N records. Consider the null pointers in the terminal nodes. An in order traversal of a B-tree will alternate between null pointers and records and will start and finish with a null pointer. There are therefore $N+1$ null pointers and a tree containing N records. From the definition of the tree, all the null pointers are at the same level; assume this is level h where the root is considered to be level 1. Thus in Figure 2.5, null pointers are at level 3. The worst case when searching the tree will require h node reads: one at level 1 through h . We can derive an expression for h in terms of N and M as follows:

At level 2 the minimum number of records is 2

At level 3 the minimum number of nodes is $2X \lceil M/2 \rceil$

.....

At level $h+1$ the minimum number of nodes is $2X \lceil M/2 \rceil^{h-1}$

(The null pointers in the terminal nodes might be regarded as pointers to nodes at a non-existent level $h+1$).

(* In the algorithm

Found: a flag to indicate if the record has been found

K : key of record being searched for

P : holds a pointer to a node

N : record count

*)

Found \rightarrow false

read root

repeat

N \rightarrow number of records in current node

case

K=key of record in current node: found \rightarrow true

K < key(R_1) : **P** \rightarrow **P**₀

K < key(R_N) : **P** \rightarrow **P**_N

otherwise : **P** \rightarrow **P** _{$i-1$} (for some i where
key(R_{i-1}) < **K** < key(R_i))

endcase

If P not null

then read node pointed to by P

until Found or P is null

Algorithm 2.1 B-tree search algorithm

We know that there are $N+1$ null pointers, and therefore

$$N+1 \geq 2X \lceil M/2 \rceil^{h-1}$$

which yields $h \leq 1 + \log_{\lceil M/2 \rceil} [(N+1)/2]$

This gives us an upper bound on the height of a tree of order M containing N records and hence an upper bound on the number of node reads during a retrieval. The minimum number of node reads is clearly one. This is the case where the record being searched for is found in the root.

Tree balancing operations are required in two cases when performing insertions or deletions on a B-tree. In the insertion operation a node can overflow because the definition of the tree imposes an upper bound on the node size. We can resolve overflow by redistributing records in the existing nodes or by splitting the overlarge node. When deleting, on the other hand, we may have node underflow because a node may become smaller than the lower bound on node size. Underflow can be resolved either by redistribution or by concatenation of two nodes [18].

2.3.4.4 B-tree insertion

New records are always inserted into a terminal node. In our diagrammatic representation of a tree, every null pointer represents an insertion point where new record might go.

Before

(... A ...)



(B ... C t D ... E)

After

(... A t F ...)

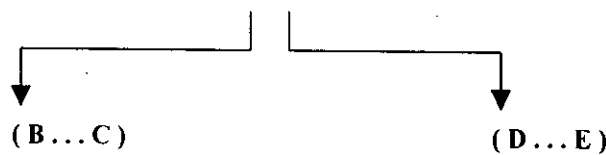


Figure 2.6 B-tree node split

In Figure 2.5, for example, records with keys greater than l but less than r would be inserted in the node containing n : to the left of n if less than n and to the right if greater than n . To determine the appropriate insertion point for a particular new record, the insertion algorithm starts by searching for the new record as it were already in the tree. The search algorithm will bring us to the appropriate point in a terminal node.

As stated earlier, a problem with inserting records is that nodes can overflow because there is an upper bound to the size of a node. What if the node into which we have inserted a record now exceeds the maximum size? The situation can be resolved using **redistribution** or **splitting**. Here, we consider how a node might split. On overflow, the node is split into three parts. The middle record is passed upward and into the parent, leaving two children behind where there was one before. Suppose that the order of B-tree

is M . The largest number of records allowable in a node is therefore $M - 1$. Splitting an overfull node with M records can be depicted as follows:

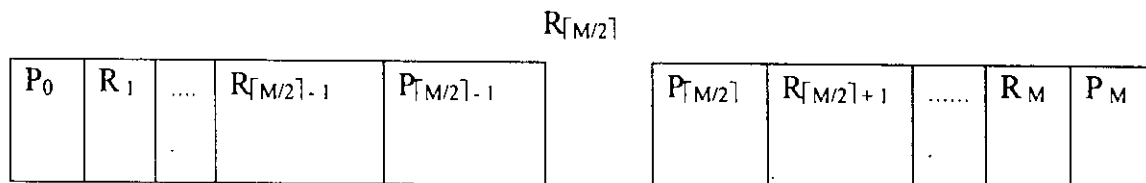


Figure 2.6 shows how a pointer to one of the two children is inserted into the parent in addition to a record. As usual, lower case letters here represents records. Uppercase letters represent pointers F is a pointer to a newly allocated node.

Splitting may propagate up the tree because the parent into which we inserted a record may have been at its maximum size. Therefore it will also split. If it becomes necessary for the root of the tree to split, then a new root is created that has just two children. This is a valid node because of third property of a B-tree. If the root splits, then the tree grows by a level. This is the only way that a B-tree grows a level. We can regard the terminal nodes as being the fixed level of a tree that grows up or down only at the top (root). Note that no explicit balancing operations are required in the insertion algorithm.

Algorithm 2.2 contains a pseudo-code algorithm for the insertion of a new record.

(* In the algorithm

In-rec : the record to be inserted into the tree
Finished : a flag to indicated if insertion has finished
Found : a flag to indicate if record has been found in the tree
P : holds a pointer to a node
TOOBIG: an oversize node
N : record count

*)

(* Search tree for In-rec forming stack of node address. *)

Found \leftarrow false

read root

repeat

N \leftarrow number of records in current node

case

key (in-rec) = key of record in current node : **Found** \leftarrow true

key (in-rec) < key(R_1) : **P** \leftarrow P_0

key (in-rec) > key(R_N) : **P** \leftarrow P_N

otherwise : **P** \leftarrow P_{i-1} (for
some i , where **key (R_{i-1}) < key (in-rec) < key(R_i)**

```

    endcase
    if P not null
        then push on to stack address of current node
        read node pointed to by P
    until found or P is null
if found
    then report record with key = key (In-rec) already in tree
else (* insert In-rec into tree *)
    P ← nil
    Finished ← false
    repeat if current node is not full
        then put In-rec and P in current node
        Finished ← true
    else copy current node to TOOBIG
        insert In-rec and p into TOOBIG
        In-rec ← center record of TOOBIG
        current node ← 1st half of TOOBIG
        get space for new node. assign address to P
        new node ← 2nd half of TOOBIG
        if stack not empty
            then pop of stack
            read node pointed to
        else (*tree grows*)
            get space for new node
            new node ← pointer to old root, In-rec and P
            Finished ← true

    until finished

```

Algorithm 2.2 B-tree insertion algorithm

The insertion algorithm assumes the existence of a stack and a temporary node, called TOOBIG, in main memory. This node has room for one more record and one more pointer than the maximum node allowed in the B-tree. It is used as temporary working space when a node splits.

The insertion algorithm starts by searching for the record to be inserted. This is done in order to bring us to the appropriate terminal node in the tree. During the search, whenever we move from a parent node to one of its children, we push the address of the parent node onto the stack. Later, this will enable us to move from a node to its parent by unstacking an address. The stack mechanism is adequate because only nodes we were interested in

are the direct ancestors of the terminal node where we start insertions. Use of the stack means there is no need for any node to contain a pointer to its parent.

An insertion of a record into the current node can have two possible results. The insertion may occur without any maintenance operations being required, or it may cause overflow.

Case 1: non overflow insertion. The current node is not full. In this case we insert the record. We also insert an appropriate pointer so that the number of pointers in the node is still one greater than the number of records. The algorithm terminates.

Case 2: overflow insertion. The current node is full. In this case we copy it into overlarge node TOOBIG, which has room for one more record and one more pointer than the maximum allowed in a tree node. We then put the records and pointers in TOOBIG back in the tree to effect the splitting operation. The center record is identified; if M is even, arbitrary choice is made between the two central records. Records and pointers to the left of the center record are put back in the current node, the remainder of which is cleared. Records and pointers to the right of the center record are put in a new node.

The center record and a pointer to the newly allocated node now have to be inserted into the parent of the current node. The algorithm therefore iterates until at some level no further splitting is needed. If the root has to split, the new root will contain, in addition to the record and pointer passed up from below, a pointer to the old root [18].

Consider the tree of Figure 2.5 and the successive insertion records with keys m , j , p and d . We will assume the tree to be of order 3. It follows that the largest node can hold two records and three pointers and that the smallest node can hold one record and two pointers. TOOBIG can hold three records and four pointers.

Insert m . This is a simple insertion.

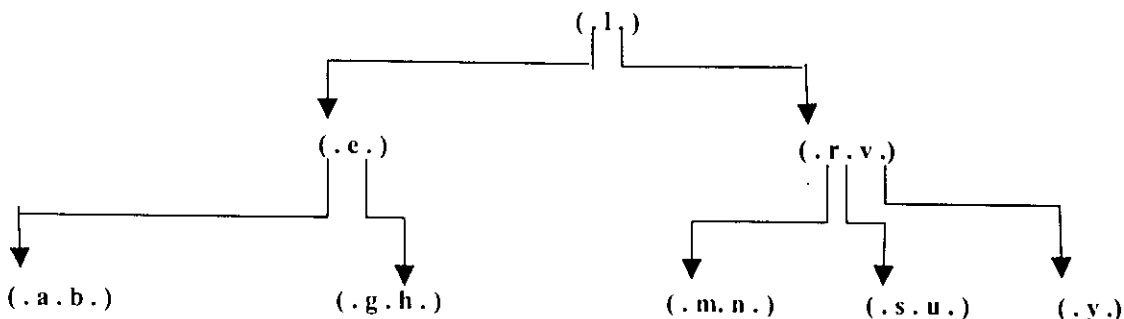


Figure 2.7 B-tree of Figure 2.5 after inserting m

The key is greater than l but less than r, so the record goes into the node with n. Because there is enough room in this node for a new record, the insertion algorithm finishes. Figure 2.7 shows the new tree.

Insert j. The record with key j should go in the node currently containing g and h. However, this node is at its maximum size, so records g, h, and j are put into the TOOBIG node. The middle record (h) is then inserted into the parent node. The remaining records form two children where there was one child before.

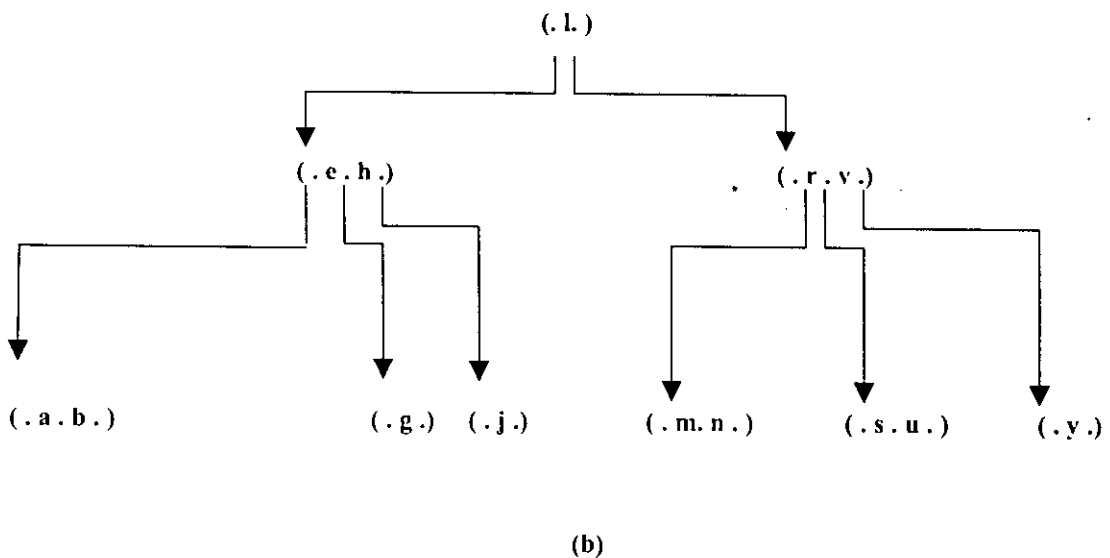
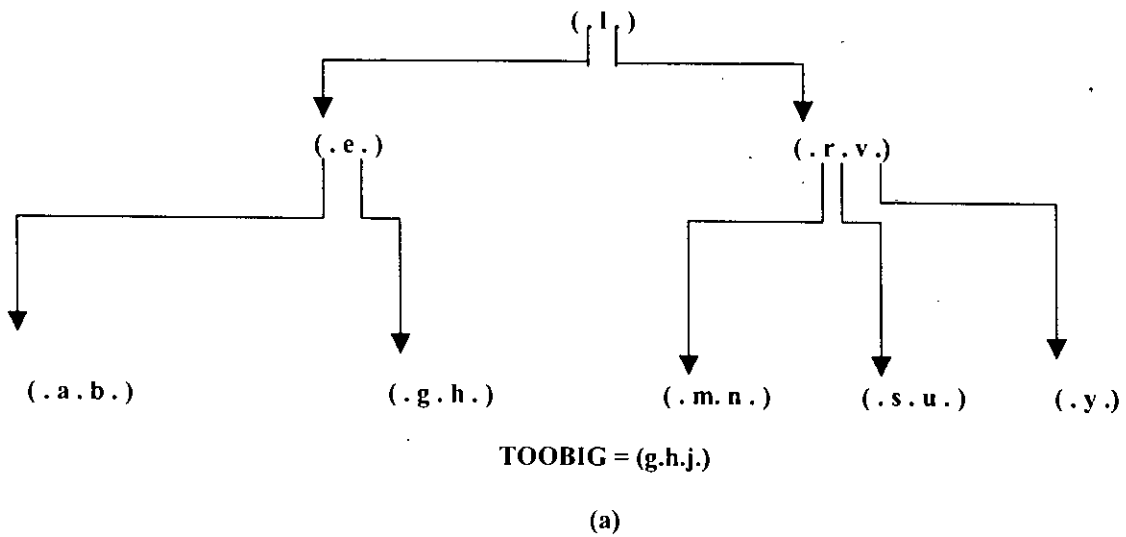
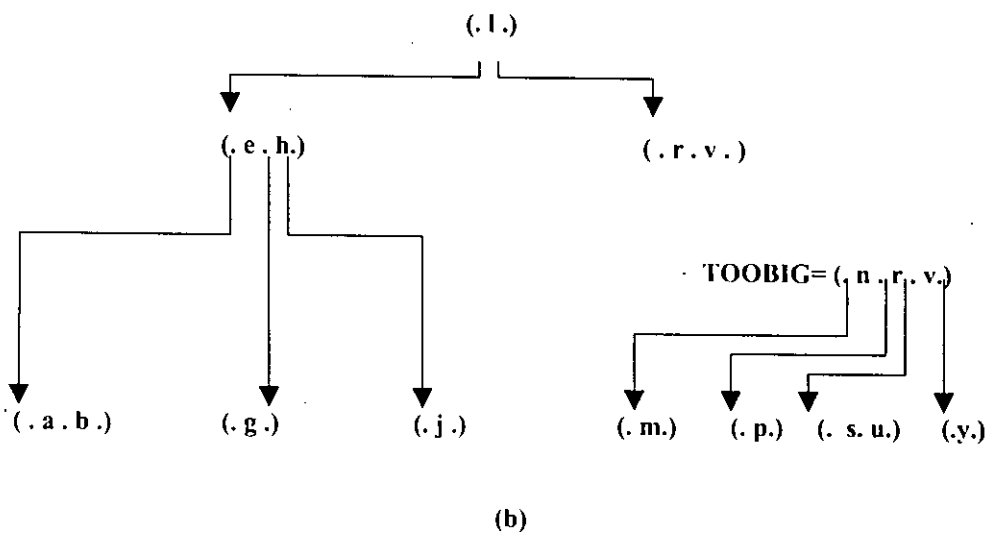
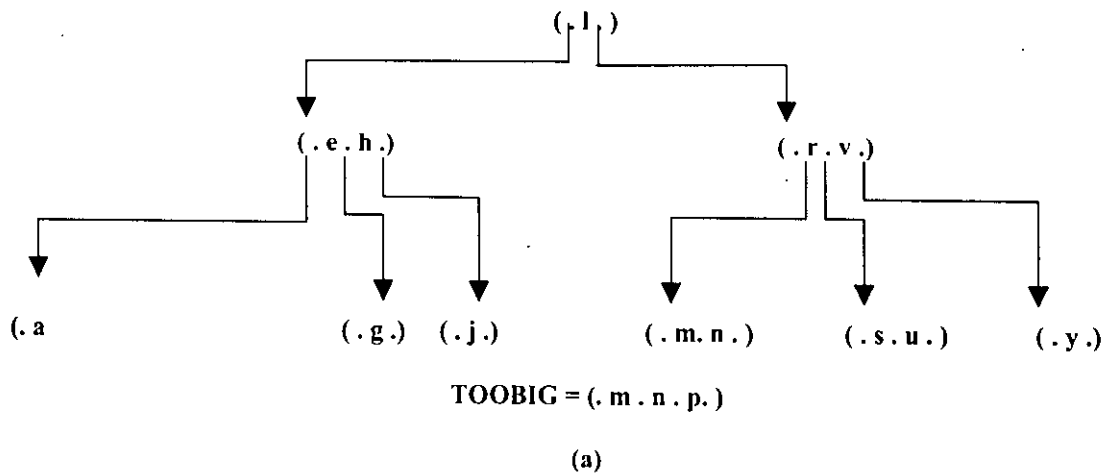


Figure 2.8 B-tree of Figure 2.9 after inserting j

Figure 2.8(a) shows the intermediate step and Figure 2.8(b) the final result of inserting a record with key j into the tree of Figure 2.7.

Insert p. When we insert a record with key p into the final tree of Figure 2.8, we find that the splitting operation occurs at two levels.

Initially the record is put in TOOBIG with records m and n (see Figure 2.9a) when this splits, record n is passed up to be inserted into the parent. However, the parent is already full, so TOOBIG is split, record r is passed up to the root. The final tree is shown in Figure 2.9(c)



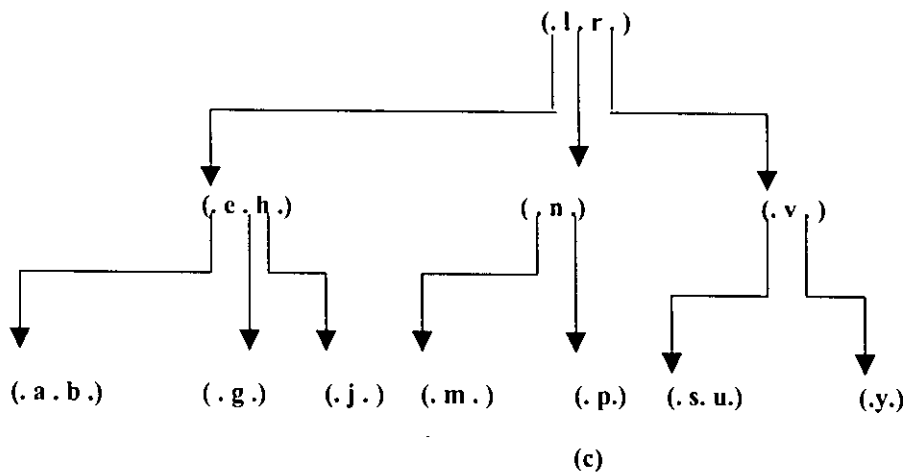
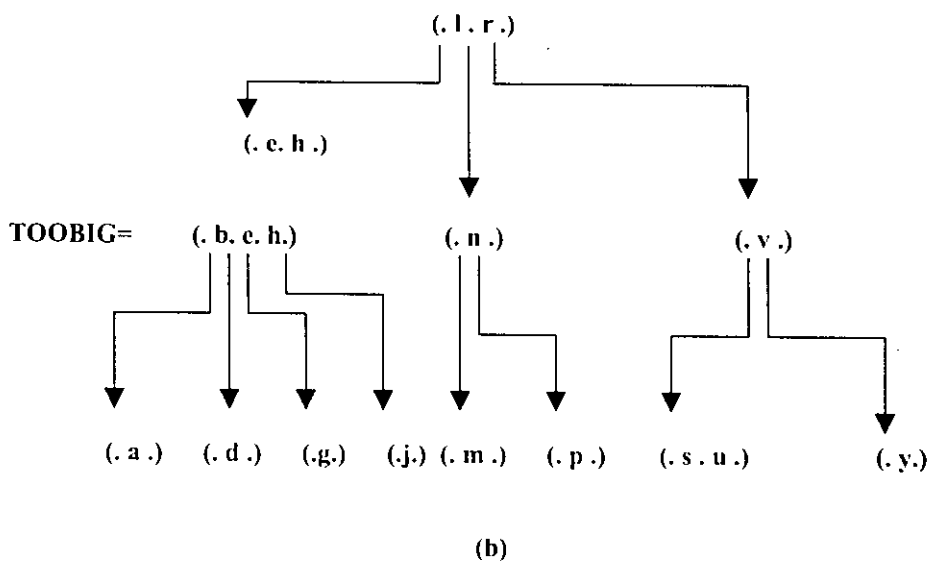
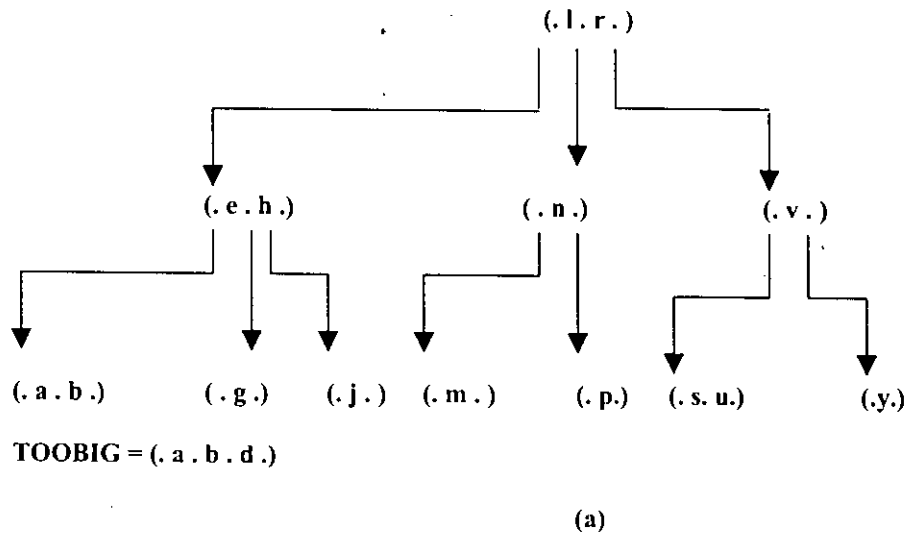


Figure 2.9 B-tree of Figure 2.8 after inserting p

Insert d. The insertion of final record into our tree causes splitting to occur all the way up to the root and the tree to grow one level.



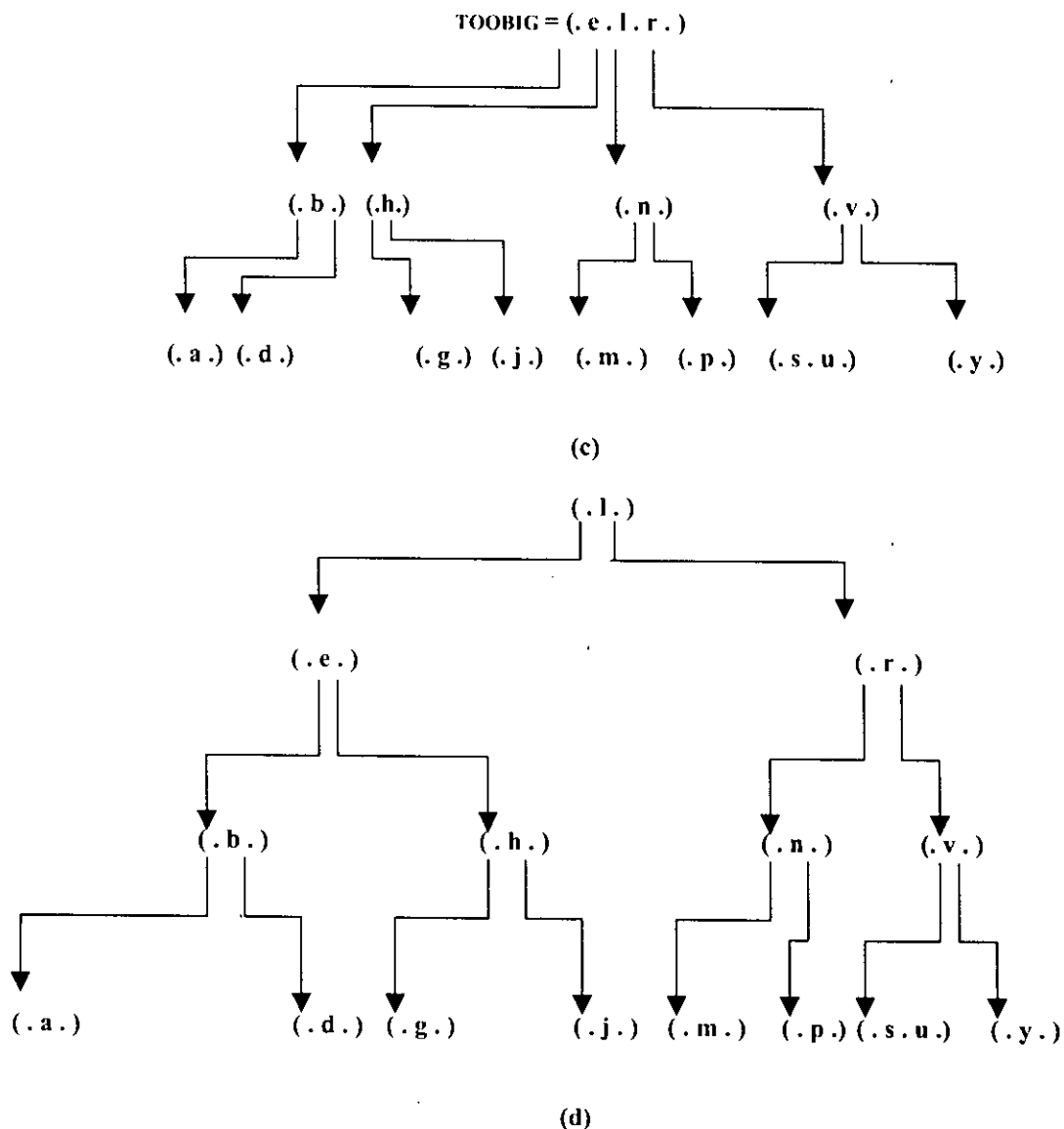


Figure 2.10 B-tree of Figure 2.9 after inserting d

Initially record d goes into TOOBIG together with records a and b (see Figure 2.10a). When TOOBIG splits, record b is passed up to the parent. However, because the parent is already full, its contents are copied with record b into TOOBIG (see Figure 2.10b). When TOOBIG splits again, record e is passed up to the root. Because the root is already full, TOOBIG is set up again containing the old root and record e (see Figure 2.10c). When TOOBIG splits for the final time a new root is created and the tree grows by a level. Figure 2.10(d) shows the final tree.

Performance of the insertion algorithm: The best case for the insertion algorithm is when there is room for the new record in the initial node. In this case we have to read h nodes

(where h is the height of the tree) and write one node. The worst case is illustrated by our last example. If the tree split all the way up to the root, then $h+1$ new nodes are created (where h is the height of the tree before insertion). That is, we must read h nodes and write $2h+1$ nodes. Knuth [13] reports that the average number of nodes split during an insertion is $1/(\lceil M/2 \rceil - 1)$, where M is the order of the tree. Thus as M increases, the average number of node splits decreases. For example, if M is 10, the expected number of splits per insertion is 0.25. This drops to .02 when M is 101. The minimum and maximum number of node reads are both h . This is because insertion is always initially into a terminal node. The minimum number of node writes is one when, as in the case of our first example, the record can be inserted in a lowest level node. The maximum number of node writes is $2h+1$, which occurs when the root splits and the tree grows a level. An alternative to splitting as a means of resolving an overlarge node is to redistribute records in a local area of the tree. If an adjacent sibling of the overlarge node has spare room, records can be moved from one node to the other. Naturally, the ordering of records in the tree must be preserved. We have not, however, included redistribution in the insertion algorithm.

2.3.4.5 B-tree deletions

As in the insert operation, we always start the delete operation at the lowest level of the tree. If the record we need to delete is not in a terminal node, then we replace it by a copy of its successor that is the record with the next highest key. The successor will be at the lowest level. We then delete the successor record. A problem with deletions is that after a record has been removed from a node we may have **underflow**; the node may be smaller than the minimum size. This situation can be resolved by means of redistribution or concatenation.

Redistribution is possible when an adjacent sibling of the node with underflow has records to spare: that is it contains more than the minimum number of records. Redistribution is possible involves moving records among the adjacent siblings and parent; thus the structure of the tree is not changed. Concatenation, which is performed when redistribution is not possible, involves the merging of nodes and is the complement of the splitting process we saw with insertions. If concatenation is performed, the structure of the tree is

changes. Changes may propagate all the way to the root. In the extreme case, the root node is removed and the shrinks by one level [18].

(*In this algorithm

Finished : a flag that indicates if deletion has finished
TWOBNODE: an oversize node that is about 50% larger than a normal node
A-sibling : an adjacent sibling node.
Out-rec : the record to be deleted from the tree.

*)

search tree for Out-rec forming stack of node addresses

Found → false

read root

repeat

N ← number of records in current node

case

key (in-rec) = key of record in current node : Found ← true

key (in-rec) < key(R_i) : **P** ← P_0

key (in-rec) > key(R_N) : **P** ← P_N

otherwise : **P** ← P_{i-1} (for some i

where key (R_{i-1}) < key (in-rec) < key (R_i))

endcase

if **P** not null

then push on to stack address of current node

read node pointed to by **P**

until found or **P** is null

if found

if Out-rec is not in terminal node

then search for successor record of Out-rec at terminal level (stacking node addresses)

copy successor over Out-rec

terminal node successor now becomes the Out-rec

(* remove record and adjust tree *)

Finished → false

repeat

remove Out-rec (record R_i) and pointer P_i

if current node is or is not too small

then Finished → true

else if redistribution possible (* an A-sibling > minimum *)

then (* redistribute *)

copy "best" A-sibling , intermediate parent record, and

current (too small) node into TWOBNODE

copy records and pointers from TWOBNODE to "best"

A-sibling, parent, and current node so A-sibling and

current node are roughly equal size.

Finished → true

```

else (* concatenate with appropriate A-sibling *)
    choose best A-sibling to concatenate with
    put in the leftmost of the current node and A-sibling the
    contents of both nodes and the intermediate record from
    the parent
    discard rightmost of the two nodes
    intermediate record in parent now becomes Out-rec
until finished
if no records in root
    then (* tree shrinks *)
        new root is the node pointed to by the current root
        discard old root

```

Algorithm 2.3 B-tree deletion algorithm

Algorithm 2.3 contains a pseudo-code algorithm for the deletion of a record from B-tree. The deletion algorithm starts by searching for the record to be deleted. The deletion algorithm starts by searching for the record to be deleted. As with the insertion algorithm, node addresses are put on a stack during the search to make it simple to move from a node to its parent later. If the record is not in a terminal node, then we can not delete it directly. Instead, we move to its successor. Because of the structure of a B-tree, the successor of any record will be at the lowest level and will be in a terminal node. The redundant lowest level record is then deleted. Thus in all cases deletion involves removing a record from a terminal node. The successor of record R_i is the first record in the sub tree pointed to by P_i . It can be located by moving down the P_0 pointers in that sub tree until the lowest level is reached.

In addition to removing the record from the current node, we also remove one of the adjacent pointers. In this way the number of records in the node will still be one less than the number of pointers. We choose, arbitrarily, to delete the pointer following the deleted record. If the new node size is not below the minimum, the algorithm terminates.

We can deal with underflow either by redistribution or by concatenation. Usually a too-small node can be resolved by redistributing records in a local area of the tree. Redistribution is possible if either adjacent sibling contains more than minimum number of records. Redistribution involves moving records from the selected adjacent sibling through the parent to the too-small node, one of its adjacent siblings, and a record from the parent. If M , the order of the tree, is odd, then the capacity of TWOBNODE must be

$(1.5M - 1.5)$ records and $(1.5M - 0.5)$ pointers.

If M is even, then the capacity must be

$(1.5M - 2)$ records and $(1.5M - 1)$ pointers.

Redistribution involves bringing into TWOBNODE the contents of the too-small node, one of its adjacent siblings, and the appropriate separating record from the parent. These records and pointers are then redistributed in a way similar to the splitting of TOOBIG in the insertion algorithm. The central record from TWOBNODE is the one written back to the parent. The left and right halves remaining are written back to the two siblings.

Given a choice of sibling nodes to use, we might reasonably choose to use the one that will cause the new sizes of the two sibling nodes to be closest to 75% full. They would then be as far as possible from the two size bounds. We could thus hope to minimize the possibility, assuming insertions and deletions to be equally likely, of future expensive splitting, concatenation, or redistribution operations. To avoid additional node reads when deciding which to use, parent nodes could hold, together with each pointer to a child, a count of the number of records the child contains. However, while this would speed up the delete operation, maintaining the counts would be a considerable overhead in the insert operation.

Redistribution is not possible if the too small node does not have an adjacent sibling node that is more than minimally full. In this case we have to use concatenation. We merge the too-small node with one of its adjacent siblings and the appropriate separating record from the parent. The resulting node replaces one of the concatenated nodes that are then discarded.

If we examine the properties of the B-tree, we see that concatenation is possible only in relatively rare circumstances. If M , the order of the tree, is odd, then the concatenation is possible only if an adjacent sibling is minimally full, that is, contains $(M - 1)/2$ records. If the siblings were larger than this, the node would exceed the maximum size after concatenation. If M is even, then concatenation is possible only if a sibling is minimally full or contains one record over the minimum. We are unlikely, therefore, to have a choice between siblings with which to concatenate. However if there is such a choice, we could again choose the sibling that results in the size of the new node being furthest from the two extremes.

Concatenation of two children removes a record from the parent; the separating record that is used in forming the new node has to be deleted from the parent node. If the parent node becomes too small by this deletion, then the problem of resolving a too-small node

has to be at the next level up. In the most extreme case, concatenation takes place all the way up the tree. It may be that we remove the only record in the root, leaving just a pointer. In this case we can discard the root; the node pointed to by the pointer becomes the new root. This is the only way in which the height of a tree can decrease.

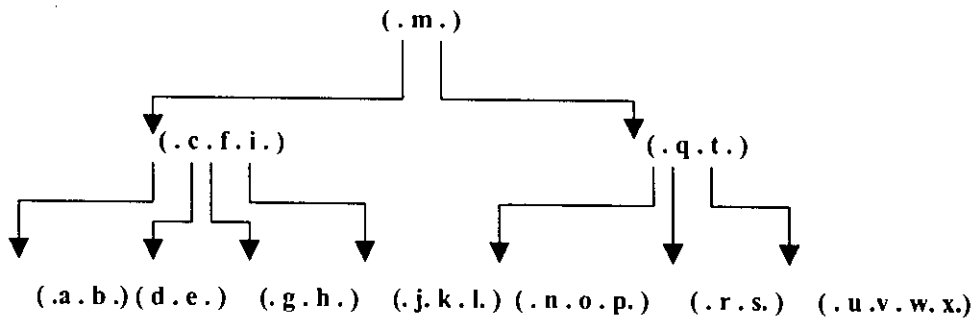


Figure 2.11 Example of B- tree

Consider the tree of Figure 2.11 and the successive deletion of records with keys *j*, *m*, *r*, *h*, and *b*. We will assume the tree of order 5; it follows that the largest node can hold four records and five pointers and smallest node two records and three pointers. TWOBNODE can hold six records and seven pointers.

Delete j. The first deletion is a simple one from the lowest level of the tree. The tree after deletion is shown in Figure. 2.12.

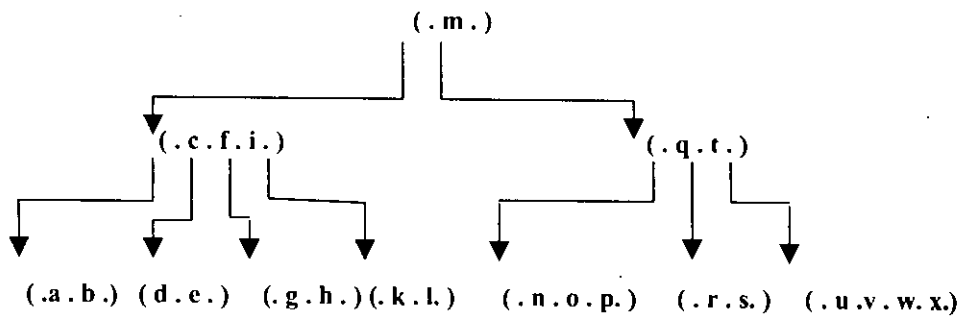


Figure 2.12 B- tree of Figure 2.11 after deleting *j*

Delete m. In this case the record to be deleted is not at the lowest level in the tree, so we replace it with a copy of its successor (the record with key *n*) and then delete the lowest level successor.

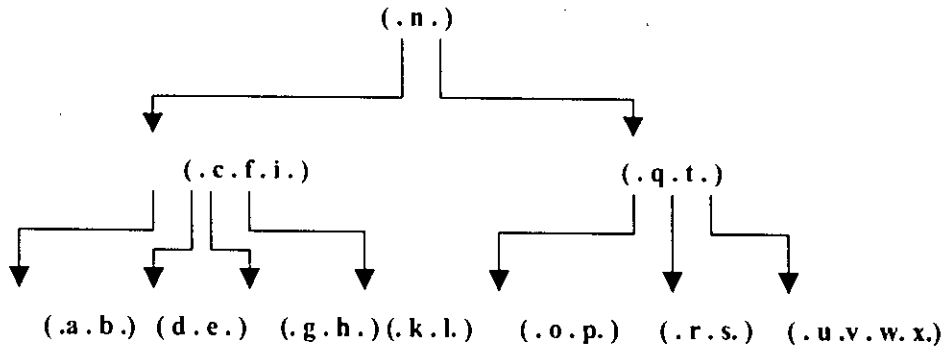
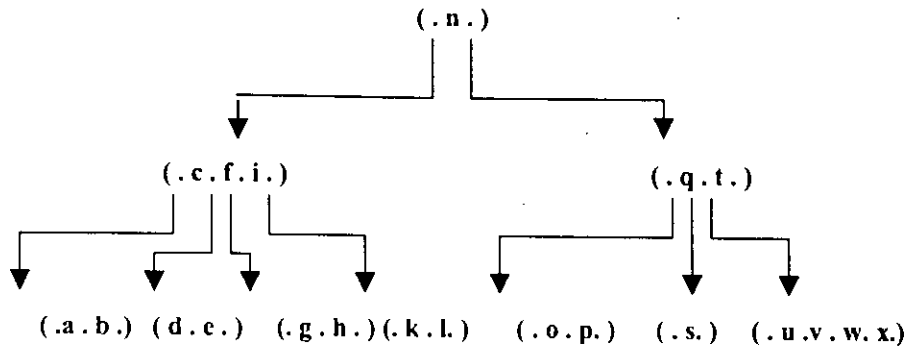


Figure 2.13 B- tree of Figure 2.12 after deleting m

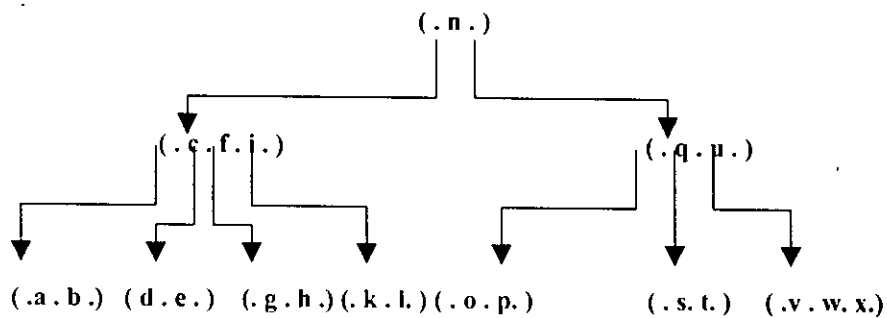
The resulting tree shown in Figure 2.13.

Delete r. Deletion of record r. Deletion of record r makes the resulting node too small. However, we can resolve the situation without altering the structure of the tree by redistributing records, because one of the adjacent siblings is more than minimally full. Records in the two adjacent siblings, into TWOBNODE and redistributed. Figure 2.14(a) shows the tree after the initial deletion and also shows the contents of TWOBNODE. Figure 2.14(b) shows the tree after redistribution.



(a)

TWOBNODE = (.s.t.u.v.w.x.)



(b)

Figure 2.14 B- tree of Figure 2.13 after deleting r

Delete h. When we delete the record with key h, the resulting node is again too small. However, in this case we can not resolve it using redistribution because neither adjacent sibling has records to spare. We therefore use concatenation. Records and pointers from the too-small node and the separating record from the parent are inserted into an adjacent sibling. The too-small node is then discarded.

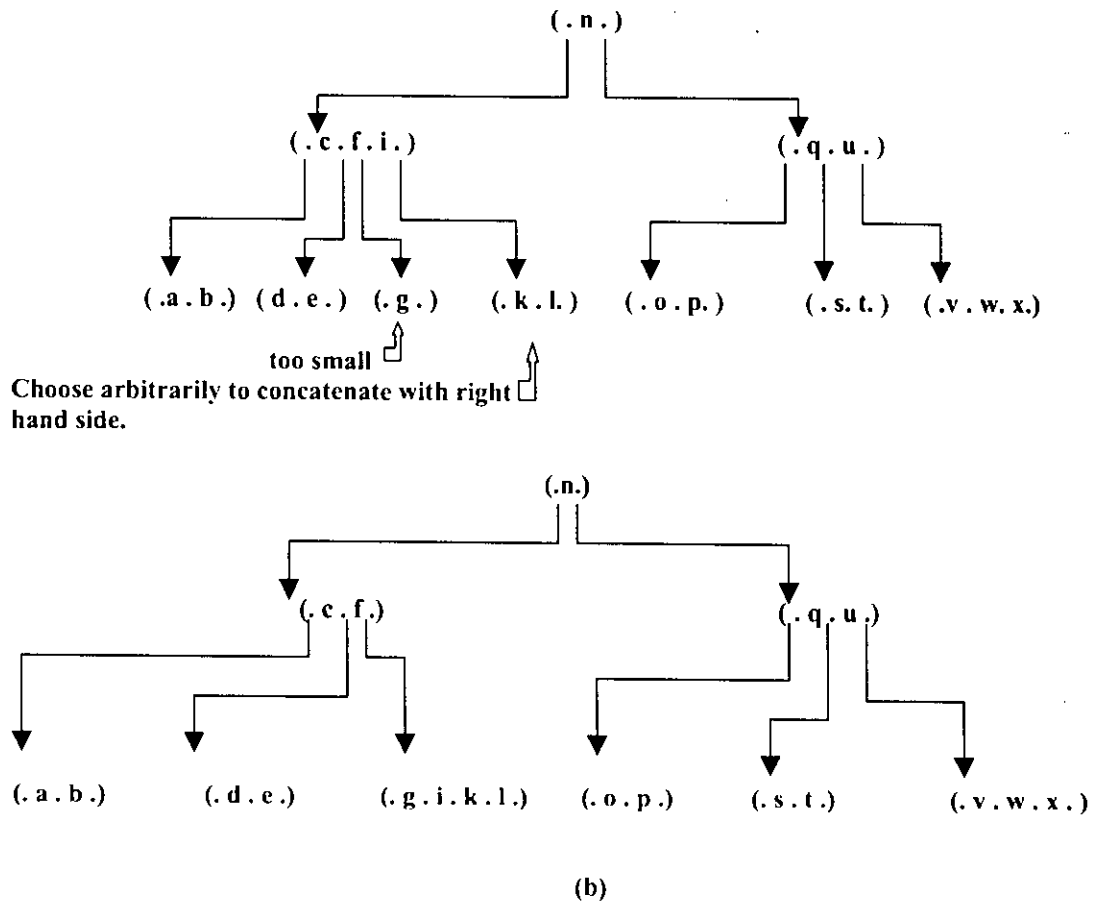


Figure 2.15 B- tree of Figure 2.14 after deleting h

Figure 2.15(a) shows the tree after the deletion and shows the choice adjacent sibling. Figure 2.15(b) shows the tree after concatenation.

Delete b. When record with key b is deleted from the tree of Figure 2.15(b), we have underflow. This is resolved using concatenation. However, in this case, removing a record from the parent causes it in turn to become too small. The too small node can not be resolved using redistribution because its only sibling is minimally full. Therefore, concatenation takes place at this level, too.

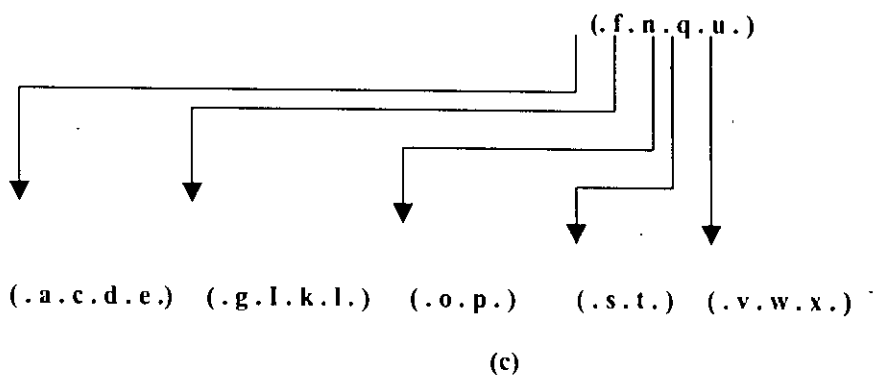
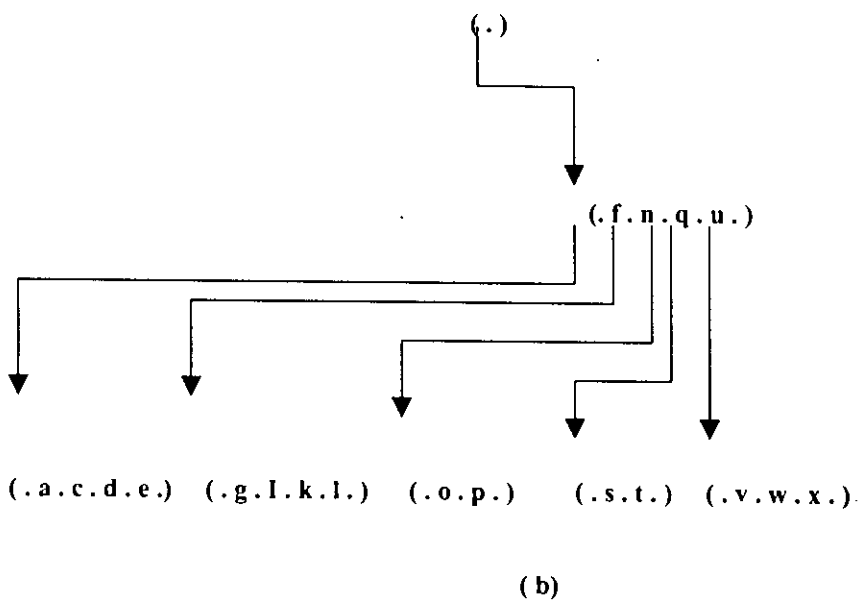
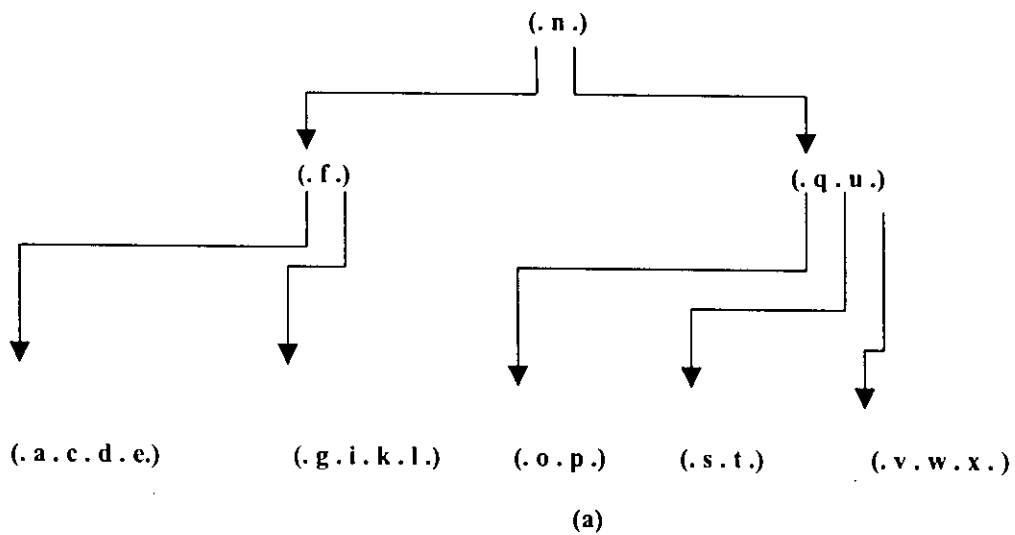


Figure 2.16 B-tree of Figure 2.15 after deleting b

The result is shown in Figure 2.16(b). Note now, however, that bringing a record down from the root causes the root to contain no records. It can therefore be removed; thus the final tree is shown in Figure 2.16(c).

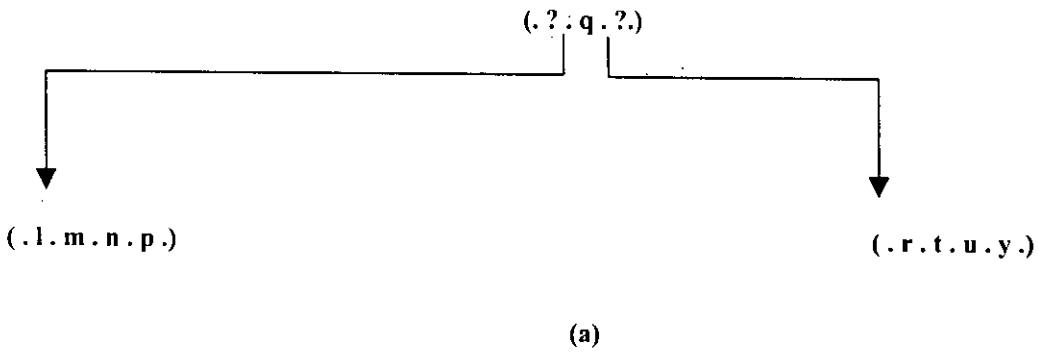
2.3.4.6 Performance of the deletion algorithm.

The best case of the deletion algorithm is illustrated by first example, that is, when the record to be deleted is at the lowest level. In this case we have to read h nodes (where h is the height of the tree) and write one node (to put back the modified node). The worst case, according to Bayer and McCreight [5], occurs when concatenation occurs at all but the first two nodes in the path from the root to the lowest-level deletion node, the child of the root has underflow, and the root itself is modified. In this case, $2h - 1$ nodes are read and $h+1$ nodes are written. However, because the majority of records are at the lowest level, Bayer and McCreight report that on average during a delete operation the number of node reads is less than $h+1+1/K$ and the number of node writes is less than $4+2/k$, where $k = \lceil M/2 \rceil - 1$.

A number of variations on the B-tree data structure have been devised. Typically, each is designed to overcome some of the deficiencies of the B-tree. In the next two sections we consider two such variations: B*-trees, which arise from a suggestion by Bayer and McCreight [5], and B⁺-trees, which were suggested by Knuth [13].

2.3.5 B*-trees

The B*-tree performs searches in the same way as the B-tree, except that there are different operation of a B-tree more efficient by reducing the number of occasions when a node had to be split. If the node into which we need to insert a record is full, we might in certain



TOOBIG = (.l.m.n.o.p.q.r.s.t.u.y.)

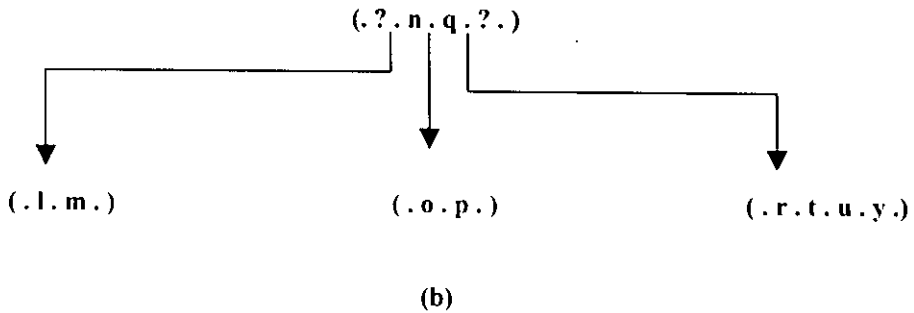


Figure 2.17 Node splitting in B*-tree

circumstances are able to solve the overflow problem by local redistribution of records rather than by splitting the nodes. Here we consider three possible techniques for redistributing records: right-only, right-or-left, and right and left. The names are derived from the adjacent sibling nodes involved.

Right-only redistribution. The right-only redistribution process is similar to the redistribution in the B-tree deletion algorithm of Algorithm 2.3. The proposed algorithm examines the right sibling of the node that is too full (or the left sibling if there is no right one). Redistribution is possible if the sibling node is not full. Thus we must split a node only if the sibling is full. When we do split, we distribute records from the two adjacent siblings (one full, one overfull) into three nodes: the two siblings and a new node. One effect of this splitting strategy is that three nodes will now be at least two-thirds full instead of at least half-full, as in a B-tree.

Right-or-left redistribution. Right-or-left redistribution is similar to right-only redistribution except when the right sibling is full. In this case, the left sibling is checked for possi-

ble redistribution. Nodes are split only when both siblings are full. Again, we distribute records from two full nodes into three nodes. Right-or-left distribution postpones node splitting, so the resulting tree will contain nodes that are, on the average, fuller than with the right- only technique.

Right-and-left redistributions. We could further than the right-or-left redistribution. At the time a node is split in the right-or-left technique we are likely to have copies of the three nodes and the two parent records in main memory. These are the originally full node, its right and left siblings, and the separating records in the parent node. At this point we could redistribute three nodes into four. With this algorithm the lower bound on node size would in most cases be raised to three-quarters full. However, not every node has two siblings, so the split routines involving the right or left sibling only would have to be employed occasionally.

Table 2.1 Redistribution costs

Required Disc I/O	Best Case	Worst Case
Right-only redistribution	r r w w	r r w w w
Right-or-left Redistribution	r r w w	r r r w w w
Right-and-left redistribution	r r w w	r r r w w w w

We can compare the expected performance characteristics of the three possible redistribution technique based on the number of reads and writes required on the B*-tree

File under the following three assumptions. First, we assume that all nodes required will be available in main memory. Second, we assume that the necessary preconditions for the technique have been satisfied. For example, in the case of the right-and-left technique, we assume that the node has both a right and left sibling. Third, we assume that each technique has a roughly equal probability of propagating splits up the tree. Thus, in this comparison we examine only the local effect of the three redistribution techniques on the sibling nodes. That is, we do not include rewriting the parent node in our matrices, as it is a

constant. Table 2.1 presents the number of reads (r) and (w) required for the three techniques.

The best case for each technique is when the right sibling is not full. Two nodes are read and then written back. The worst case for the right-or-left technique, for example, is when the right sibling is full and thus the left sibling must be read. The left sibling is full, so we have to split and write three nodes. Recall for comparison that a split-on-overflow strategy results in a " $r w w$ " case with nodes at least half full. What, then are the advantages of local redistribution? There are two. First, the nodes are used more efficiently with a minimum capacity of $\lfloor (2M-2)/3 \rfloor$ records in the case of the first two techniques and $\lfloor (3M-3)/4 \rfloor$ records in the third case. Second, with redistribution, splitting does not propagate up the tree. Note, however, that the range of node size in a B^* -tree is smaller than that in a B -tree of the same order. If the tree is volatile, there may consequently be more occasions on which underflow or overflow has to be resolved.

With these redistribution assumptions and the analysis above, the right-only redistribution technique seems preferable. The right-only technique is simple and gives the advantages of redistribution with little I/O overhead. Therefore, our further discussion will assume a right-only redistribution algorithm.

For an example of a case where node-splitting is necessary, assume that we have a B^* -tree of order 5. With the three nodes depicted in Figure 2.17(a). An attempt to insert a record with key o into the B^* -tree will result in a merging and splitting sequence that transforms the tree in Figure 2.17(a) into the tree in Figure 2.17(b).

The root node has no siblings. What happens if it needs to split? As before, the central record will become the new root and the remaining parts of the old root will form the first level of the tree. However, to ensure that these children are not smaller than the new minimum size we defined above, the upper bound on the root has to be modified. For a B^* -tree of order M the upper bound of the root node will now be $2\lfloor (2M-2)/3 \rfloor$ records. When the roots splits, it will leave two nodes each containing $\lfloor (2M-2)/3 \rfloor$ records. Thus we now have a tree with two different node capacities (root node and other nodes).

Knuth [13] termed the tree that results from these modifications a B*-tree of order M has the following properties:

1. No node apart from the root has more than M children.
2. Every node, except for the root and the terminal nodes, has at least $\lfloor (2M-2)/3 \rfloor + 1$ children.
3. The root unless the tree has only one node, has at least two children and at most $2\lfloor (2M-2)/3 \rfloor + 1$ children.
4. All the terminal nodes appear on the same level, that is, they are the same distance from the root.
5. A terminal node with K children contains K - 1 records. A terminal contains at least $\lfloor (2M-2)/3 \rfloor$ records and at most M - 1 records.

Table 2.2 Maximum and minimum node sizes

		M = 20	M=21	M=22
B-tree (order M)				
Minimum records (root)	1	1	1	1
Maximum records (root)	M - 1	19	20	21
Minimum records (non-root)	$\lceil (M - 2)/2 \rceil$	9	10	10
Maximum records (non-root)	M - 1	19	20	21
B*-tree (order M)				
Minimum records (root)	1	1	1	1
Maximum records (root)	$2\lfloor (2M-2)/3 \rfloor$	24	26	28
Minimum records (non-root)	$\lfloor (2M-2)/3 \rfloor$	12	13	14
Maximum records (non-root)	M - 1	19	20	21

Table 2.2 shows maximum and minimum node sizes for root and non-root nodes. The Figures for a B-tree and a B*-tree are given for trees of order 20, 21, and 22.



Deletions. What we do if a deletion leaves a node too small? As in the case of a B-tree, we can locally redistribute records if an adjacent sibling has records to spare. If there is no such sibling we must concatenate. Note that only concatenation normally allowable is of three nodes into two. (The exception is when the only two children of the root are concatenated and the tree shrinks a level.). What happens, however, if we are dealing with a node at one end of a tree level? Such a node has only one adjacent sibling. See for instance, node X in Figure 2.18(a) or (b). We know that node W is minimally full; otherwise redistribution would have been possible. However, we can not necessarily concatenate V, W, and X. If V is not minimally full, the result will be too large to fit into two nodes. One solution is to redistribute a single record so that W is now too small and X is minimally full. Now we have an instance of the more general case. Redistribution can be tried; if it turns out that V is minimally full, then V, X, and W can be concatenated.

Comparison with B-tree. The height (h) of a B*-tree of order M containing N records is given in the following expression:

$$h \leq 1 + \log_{(2M-1)/3} [(N+1)/2]$$

Searching a B*-tree is faster on average than searching a B-tree for a particular N and M due to the higher average branching factor. However, B-trees are better for applications where insertions and deletions are more common than searches. B*-trees are thus better when searching is the most common tree operation.

2.3.6 B⁺-trees

Knuth proposed a variation on B trees that for clarity Comer designed the B⁺-tree. Records in a B⁺-tree are held only in the terminal nodes of the tree. The terminal nodes are linked together to facilitate sequential processing of the records and termed the sequence set. Non-terminal nodes are indexes to lower levels in a similar way to the structure of Figure 2.1. Nodes in the index levels contain only key values and tree pointers. There is no need for terminal nodes tree pointer fields. Thus terminal nodes have a different structure from non-terminal nodes. In fact there is no reason why the index part of the tree should not be stored on a different device than terminal nodes.

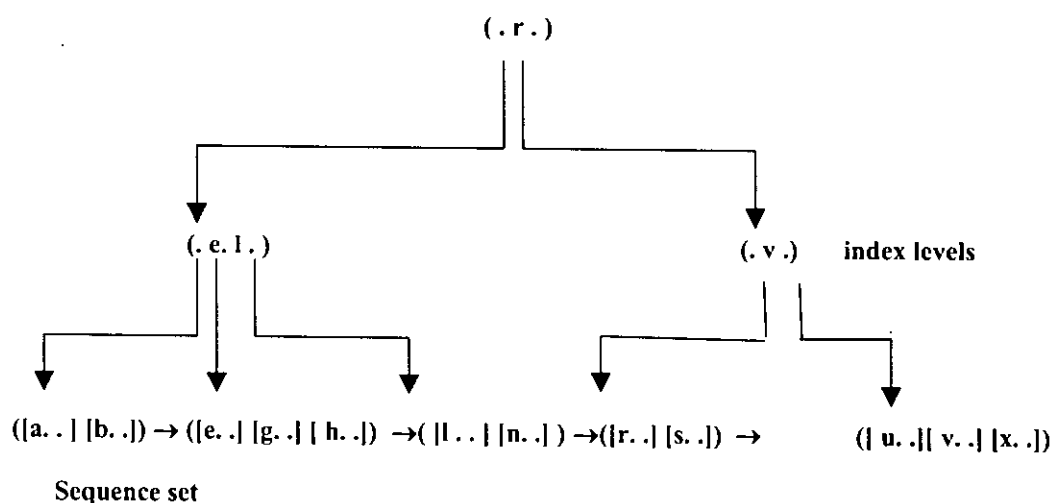


Figure 2.19 Example of B⁺-tree

Figure 2.19 shows a B⁺-tree. To make distinction clear, index records in the tree are shown thus:

[key...]

and keys are shown as dingle letters. The records in tree are those of the B-tree of Figure 2.5 . The records were inserted into the B⁺-tree in arbitrary order; thus the index structure shown is one of many possibilities.

Recall that terminal nodes contain records and index level nodes contain only keys. In addition, for efficiency, node capacity is likely to be a function of the physical record size. Therefore it is likely to be a function of the physical record size. Therefore it is likely that the order of a B⁺-tree index will be different from the capacity of the terminal nodes in its sequence set. Suppose, for example, that physical records are 512 bytes long that index records are composed of an 8-byte key and a four byte pointer to the file being indexed, and that internal tree pointers occupy 2 bytes. We can pack 42 index records in a 512 bytes terminal node and leave 8 bytes for pointer to the next terminal node. In the index levels we can have 50 eight-byte keys and 51 tree pointers in each 512- byte node and have 10 bytes free in which to store the number of records currently in the node. The properties of a B⁺-tree of order M are as follows:

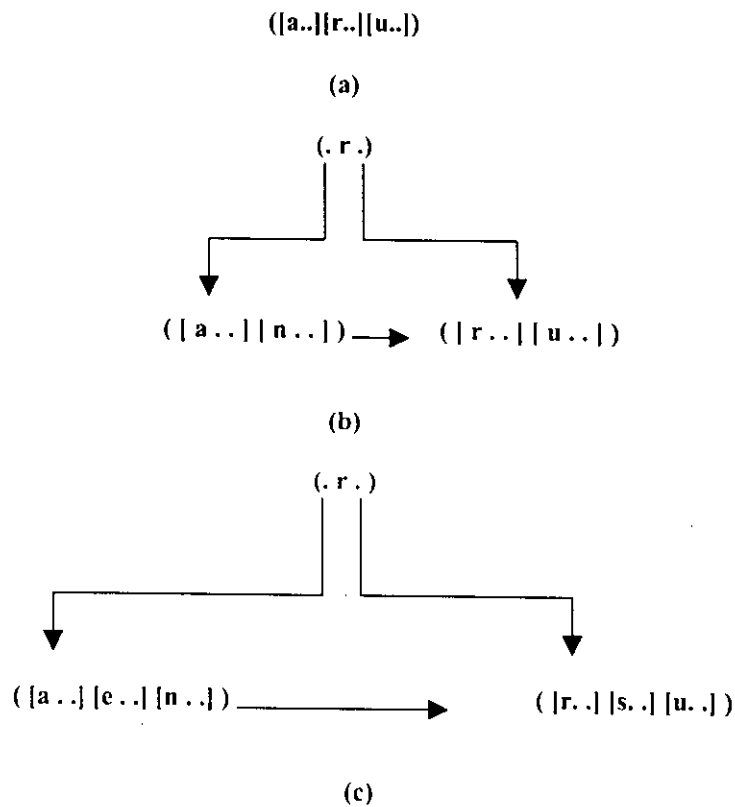
1. The root node has 0, 2, or $\lceil M/2 \rceil$ through M children.
2. All nodes except the root and terminal nodes have at least $\lceil M/2 \rceil$ and not more than M children.

3. All terminal nodes appear at the same level; that is, they are the same distance from the root.
4. A terminal node with K children contains K - 1 keys.
5. Terminal nodes represents the sequence set of the data file and are linked together.

2.3.6.1 Insertions

If the node splits when we insert a record into a terminal node of a B⁺-tree, we put a copy of the key of the central record in TOOBIG into the index. We then divide all the records in TOOBIG between the old node and a new node. Thus the central record will also be in one of the two halves after splitting. If an index node has to split, the algorithm is the same as for a conventional B-tree and the central record is passed up to the parent. Figure 2.20 shows the evolution of B⁺-tree. It is the only one of the possible ways in which the tree of Figure 2.19 might have evolved. For this example we assume that terminal nodes can hold two or three records and index node one or two keys.

Evolution of B⁺-tree



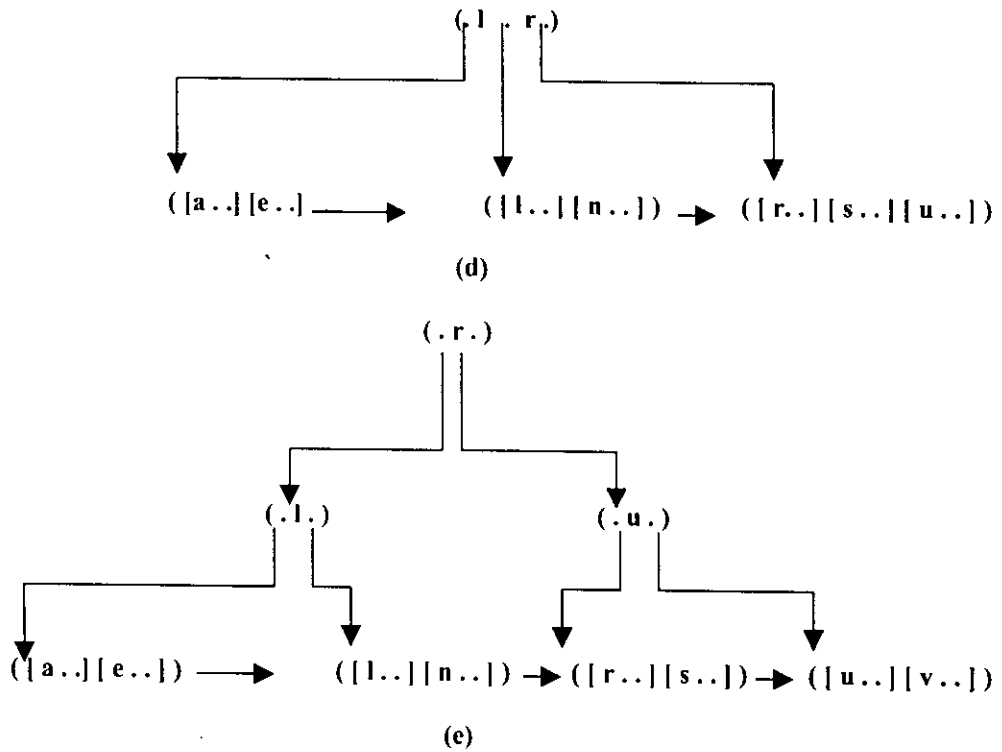


Figure 2.20 Evolution of B⁺ Tree

Figure 2.20(a) shows the tree after only three records have been inserted. Insertion of a record with key n causes the terminal node to split in two and the key r to be passed into the index. It is the first key in the index. This is shown in Figure 2.20(b). Figure 2.20 (c) shows the tree after records with keys e and s have been inserted. Insertion of a record with key l causes another terminal node to split, and a second key to be inserted into the index. This shown in Figure 2.20(d) . Finally, insertion of a record with key v causes a terminal node to split, a key to be inserted into the index, and the root of the index to split. Now we have two index levels, as shown in Figure 2.20 (e).

2.3.6.2 Deletions

When a record is deleted from the B+-tree and no distribution or concatenation is needed, no changes need be made to the index. Even if the key of the record to be deleted appears in the index, it can be left as a separator.

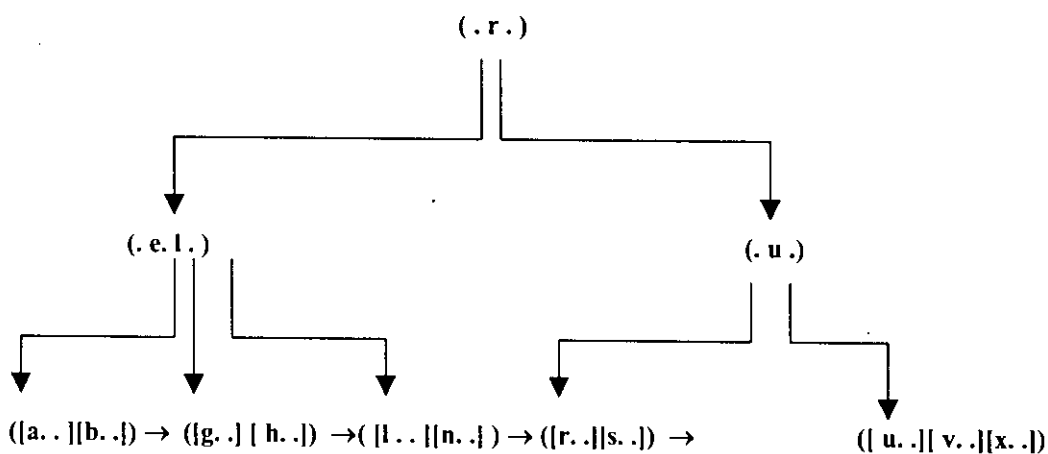


Figure 2.21 B⁺-tree of Figure 2.19 after deleting e

Figure 2.21 shows the tree of Figure 2.19 after the deletion of record with key e. Deletions that result in redistribution of records cause changes in the content but not the structure of the index levels.

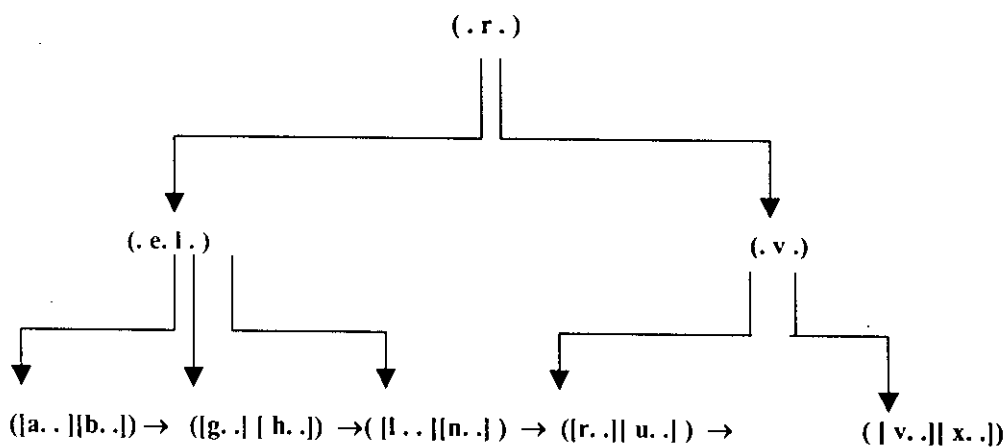


Figure 2.22 B⁺-tree of Figure 2.21 after deleting s

Figure 2.22 shows the tree of Figure 2.21 after the deletion of record with key s.

Finally deletions that result in concatenation of terminal nodes also cause deletions from the index levels.

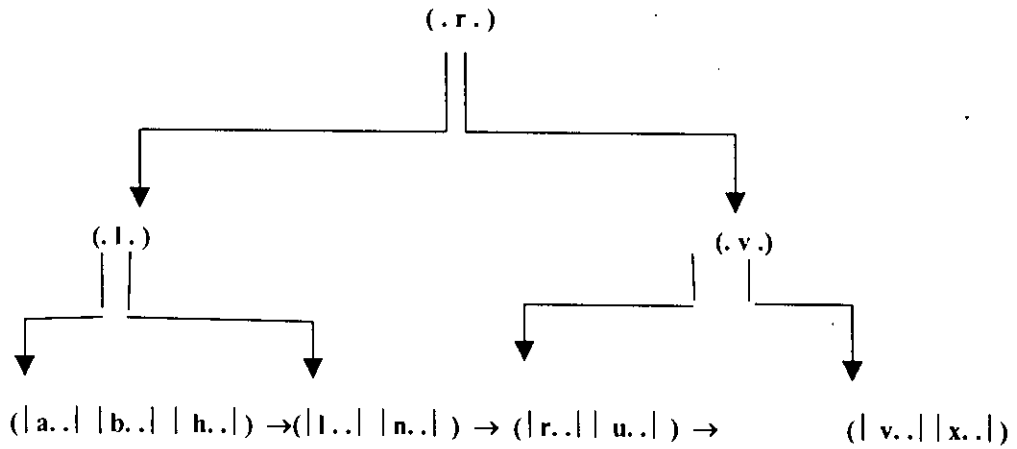


Figure 2.23 B+-tree of Figure 2.22 after deleting g

Figure 2.23 shows the effect of deleting a record with key g from the tree of Figure 2.22.

2.3.6.3 Comparison with B-tree

All searches in a B+-tree have to go down to the terminal nodes. However, because the index levels hold only keys rather than complete index records, searching time is comparable with a B-tree holding the same number of records. Sequential processing in a conventional B-tree is more complex than traversing a linked list and may also require that more than one node be held in main memory simultaneously. In a B+-tree, getting the next record in sequence requires at most one node read. B+-trees are therefore good in applications in which both direct and sequential processing are required.

2.4 Comparison of Static and Dynamic Indexes :

Held and Stonebraker compared the properties of static index structures typified by ISAM with those of dynamic structures typified by B-trees and VSAM. They suggest that while a dynamic structure is easier to reorganize, there is a price to pay for this and that the costs are threefold.

First, there may be pointers into the B-tree from other files. For example, the B-tree may be the main file with index files pointing into it. Certain operations on a B-

tree may require a record to be moved from one node to another. Such movements are likely to require changes to the pointers to the records that are moved. Thus the insert and delete operations on the B-tree may have hidden overheads.

Second, there may be concurrency problems in multi-user systems. One user may try to access a B-tree while another is updating it. The problem of locking out nodes is trivial.

Finally, B-trees need explicit pointers in non-leaf nodes because nodes can be split dynamically. A B-tree node can therefore hold fewer records than a node of the same total size that does not require such pointers. The branching factor is therefore smaller and the height of the tree likely to be greater than that of a static structure holding the same number of records. Operations such as search, insert, and delete will therefore tend to take longer than they would for a static index to a file with no overflows.

EXPERIMENTAL RESULT ON PERFORMANCE STUDY OF B, B AND B+ TREES*

3.1 Introduction

In chapter 2 we have discussed index techniques and their uses in the databases. Indices are associated with the main data file. They facilitate the Database Management Systems to access records in the data file faster and in a random fashion. There are different types of index structures and algorithms. Each has some advantages and disadvantages over others. One structure may be suitable in one context but may not be suitable in another context. We focused our discussion on dynamic index. B, B* and B+ trees are often used for implementing dynamic index. We have developed a simulation program to see the performance of these trees in different context. We have shown the results of the program in this chapter.

3.2 Simulation Program

In this thesis, we did a simulation program for constructing and using B, B* and B+ trees for database indexing. This program generates random data to insert, search and delete. Every time, before the execution, the program can take how many records need to be inserted, searched and deleted. It can select either B or B* or B+ tree and the order of the tree. Performance of each tree is measured in terms of number of comparisons, number of nodes, number of splits and the height of the tree required. At first, each tree was constructed with an adequate number of records to make it matured enough. Performance statistics were also kept during the construction. Insert, search and delete algorithms were tested only on the matured trees. Performance statistics were kept during each test. To get a good and reliable statistics for a particular test the program were executed a large number times. The results of these executions were recorded and plotted separately.

3.3 Results

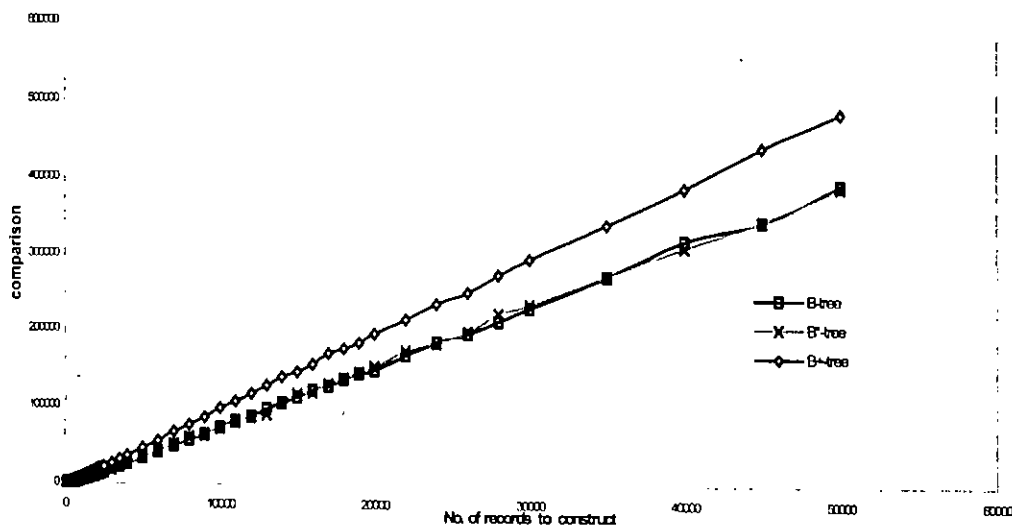


Figure 3.1: No. of Comparisons required with no. of records for constructing B, B*, B+ Trees

Figure 3.1 shows the number of comparison required with the number of records for constructing B, B*, B+ Trees. Number of comparisons increases with the number records in each tree. B, B* have the same performance in this regard and B+ has the worst performance in this regard.

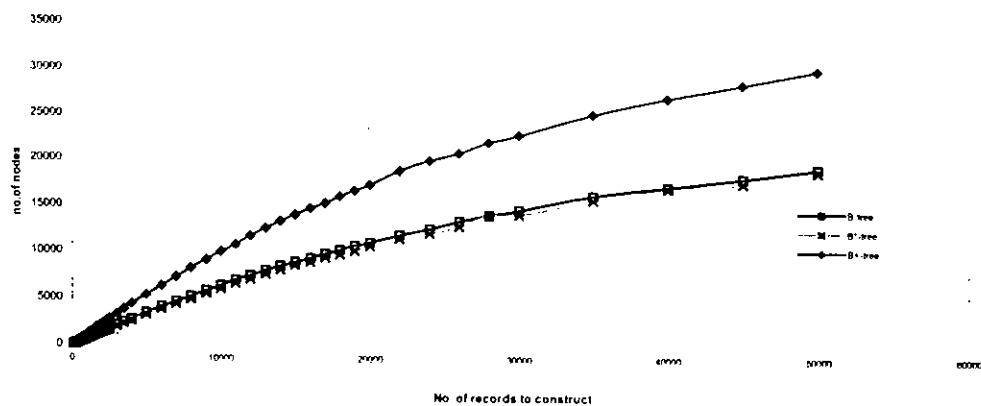


Figure 3.2: No. of Nodes required with no. of records for constructing B, B*, B+ Trees

Figure 3.2 shows the number of nodes required with the number of records for constructing B, B*, B+ Trees. B-tree and B* tree have the same performance in this regard. B+ tree is the worst since it needs the highest number of nodes.

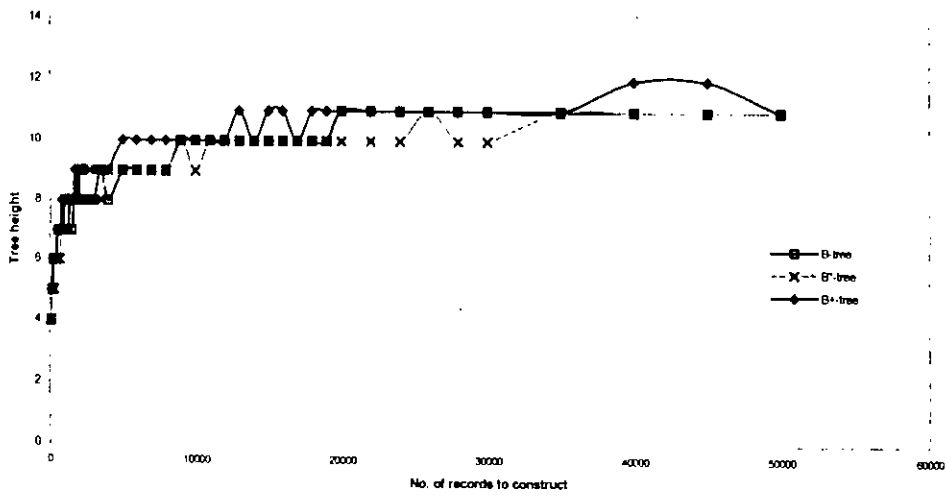


Figure 3.3: Height required with no. of records for constructing B, B*, B+ Trees

Figure 3.3 shows the height of the tree required with the number of records for constructing B, B*, B+ Trees. B+ tree has higher tree height than B tree and B* tree for a large number of records. For all the trees height increases as the number of records increases.

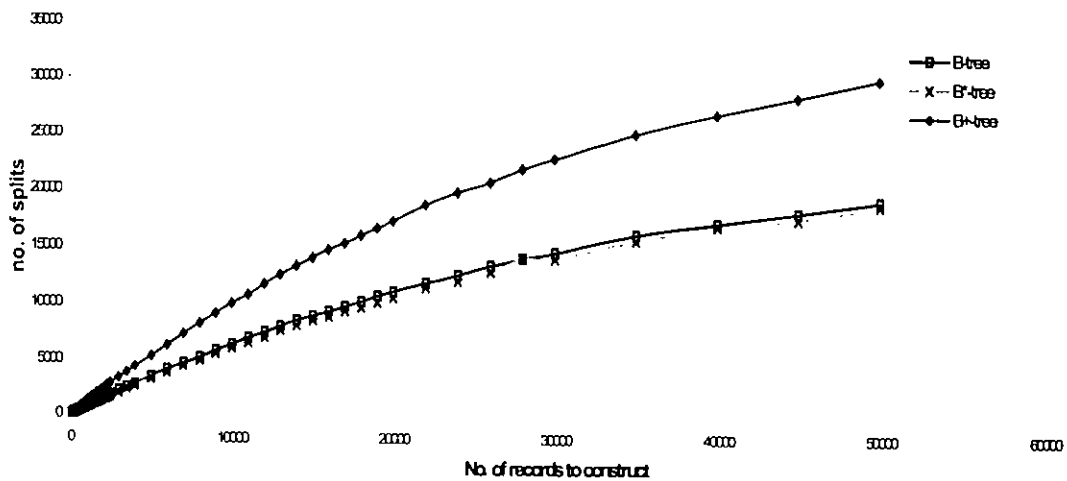


Figure 3.4: No. of splits required with no. of records for constructing B, B*, B+ Trees

Figure 3.4 shows the number of splits required with the number of records for constructing B, B*, B+ Trees. B tree and B* tree require the same number of splits. B+ is the worst since it needs the maximum number of splits. For all the trees, number of splits increases as the number of records increases.

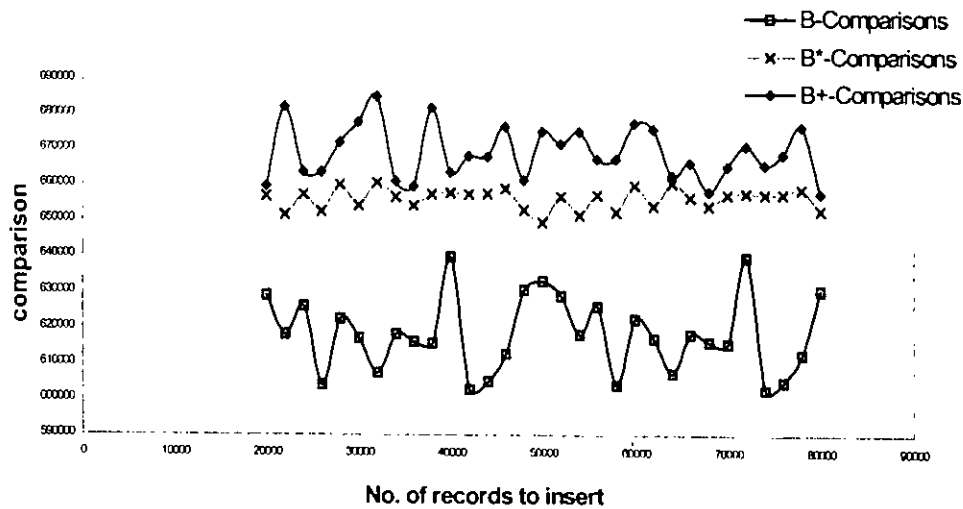


Figure 3.5: No. of comparisons required with no. of records for inserting in B, B*, B+ Trees

Figure 3.5 shows the number of comparisons required with the number of records for inserting in B, B*, B+ Trees. B tree is the best since it needs the lowest number of comparisons. B* tree has average performance. B+ tree has the worst performance it requires the largest number of comparisons than the other trees.

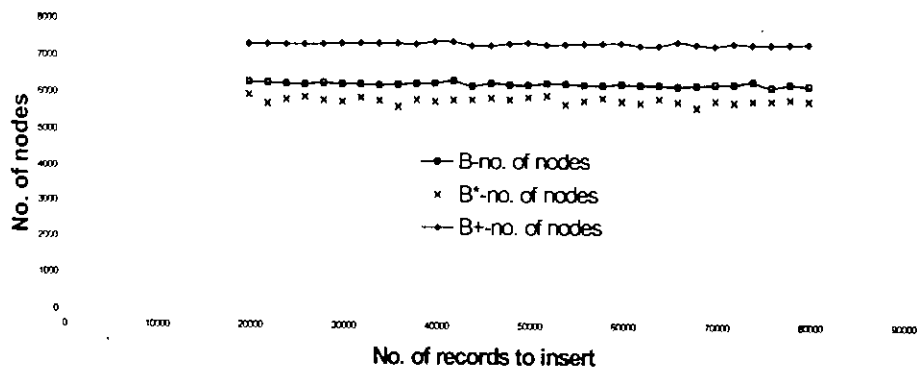


Figure 3.6: No. of Nodes required with no. of records for inserting in B, B*, B+ Trees

Figure 3.6 shows the number of nodes required with the number of records for inserting in B, B*, B+ Trees. B* tree needs few nodes for inserting any number of records. B tree has average performance in this regard. B+ tree always needs more nodes than the other trees for inserting any number of records.

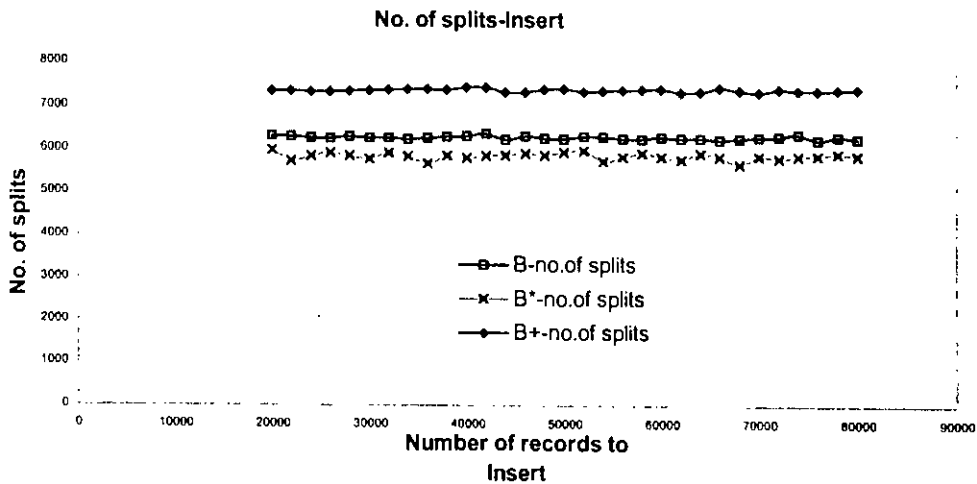


Figure 3.7: No. of splits required with no. of records for inserting in B, B*, B+ Trees

Figure 3.7 shows the number of splits required with the number of records for inserting in B, B*, B+ Trees. B* tree needs few splits for inserting any number of records. B tree needs more splits than B* tree for inserting records. B+ tree is the worst in this respect because it requires a large number of splits for inserting records.

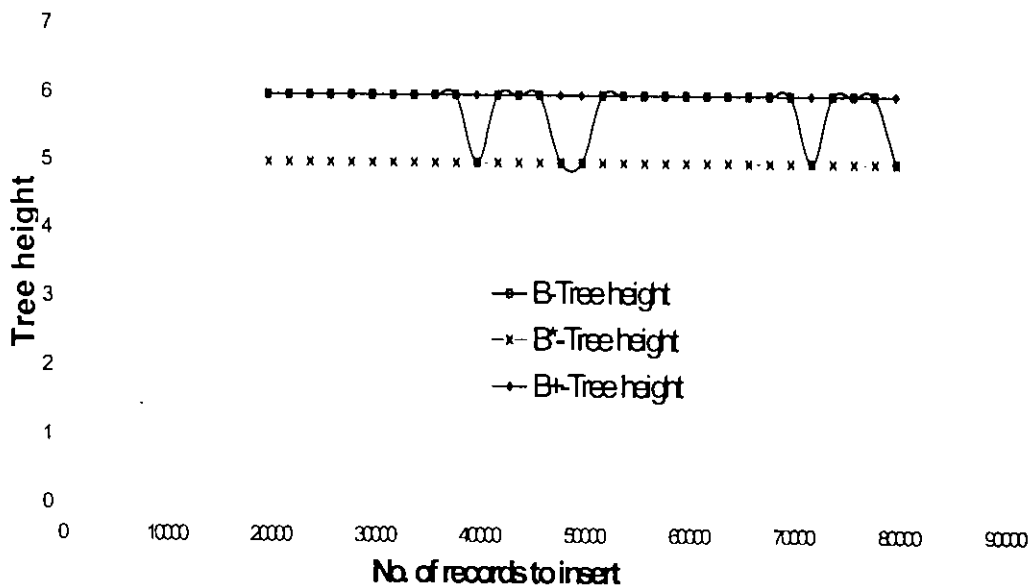


Figure 3.8: The height of the tree required with no. of records for inserting in B, B*, B+ Trees

Figure 3.8 shows the height of the tree required with the number of records for inserting in B, B*, B+ trees. B* tree always needs a small height for inserting records and B+ tree needs the maximum height for inserting records.

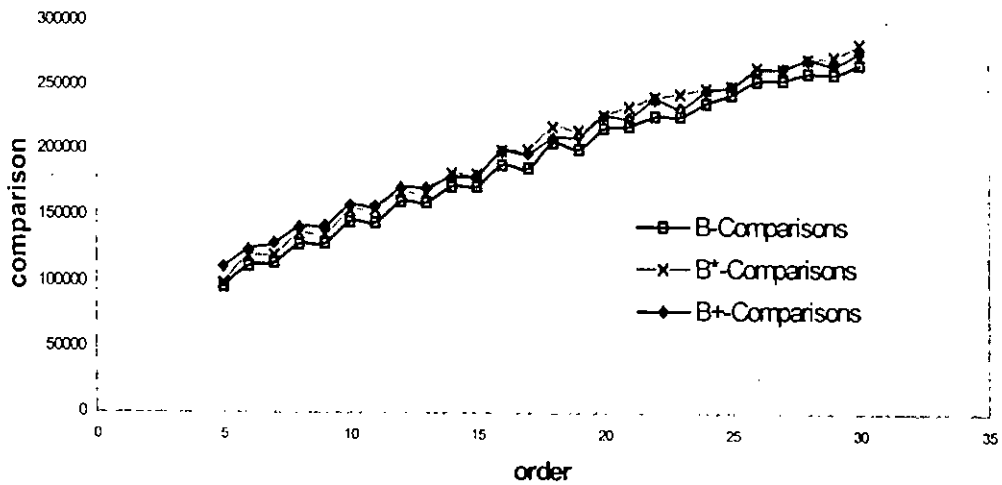


Figure 3.9: The no. of comparison required with the order of the tree for B, B*, B+ Trees

Figure 3.9 shows the number of comparisons required with the order of the tree for B, B*, B+ trees. Number of comparisons increases for all trees as order of the tree increases. B tree needs a small number of comparisons than B* and B+ trees for any order. B* and B+ trees have the same performance in this regard.

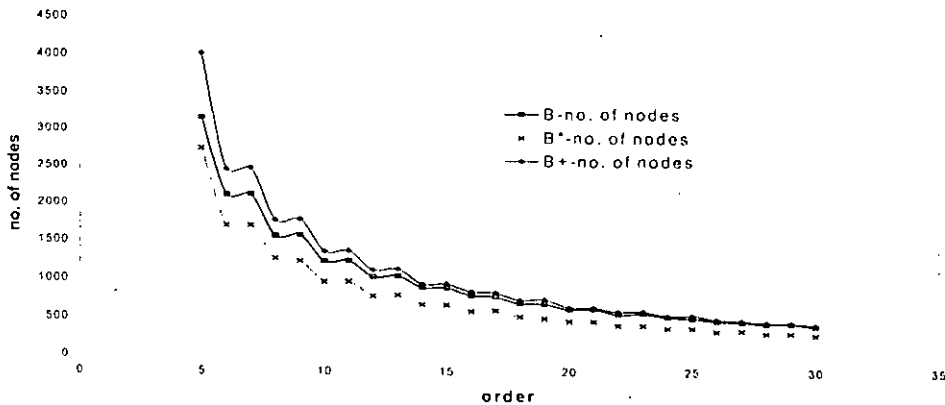


Figure 3.10: The no. of nodes required with the order of the tree for B, B*, B+ Trees

Figure 3.10 shows the number of nodes required with the order of the tree for B, B*, B+ trees. B* is the best since it needs few nodes for any order. B has average performance. B+ is the worst since it needs the highest number of nodes for any order.

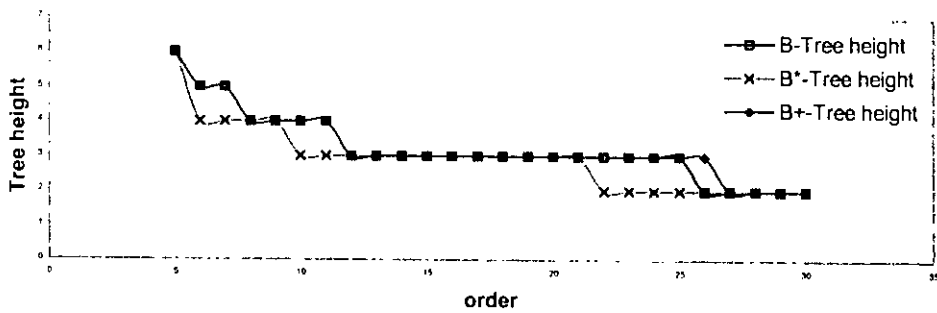


Figure 3.11: The height of the tree required with the order of the tree for B, B*, B+ Trees

Figure 3.11 shows the height of the tree required with the order of the tree for B, B*, B+ trees. B* tree is the best since its height is the minimum for any order. B has average performance. B+ tree is the worst since its height is the maximum for any order. For all trees height decreases as order increases.

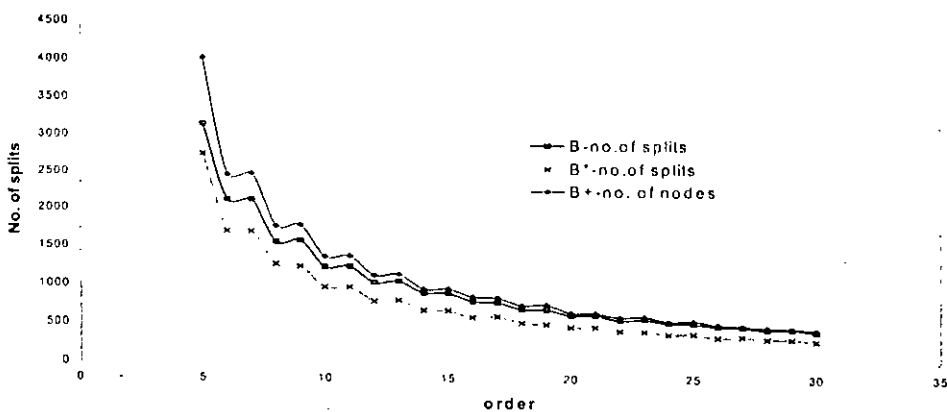


Figure 3.12: The no. of splits required with the order of the tree for B, B*, B+ Trees

Figure 3.12 shows the number of splits required with the order of the tree for B, B*, B+ trees. B* tree is the best since it needs the minimum number of splits for any order. B has average performance. B+ tree is the worst since it needs the maximum number of splits

for any order. For all trees number of splits decreases as order increases

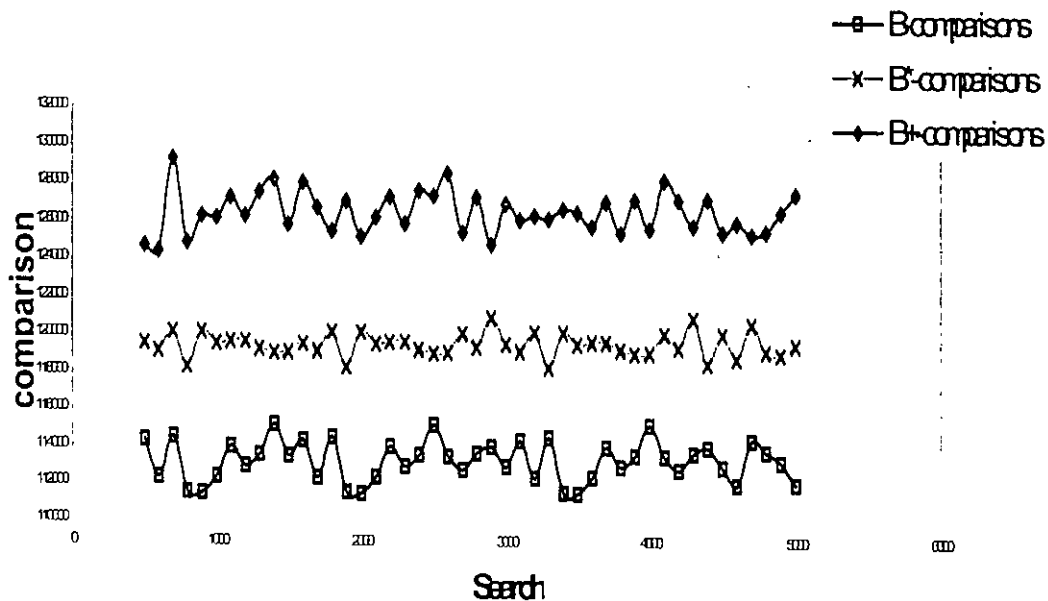


Figure 3.13: The no. of comparisons required with the no. of records searched in B, B*, B+ Trees

Figure 3.13 shows the number of comparisons required with the number of records searched in B, B*, B+ trees. B tree is the best since it needs few comparisons for any number of record searching. B* tree has average performance. B+ tree has the worst performance since it needs more comparisons for any number of record searching.

3.4 Conclusion

Theoretically, B trees are better for applications where insertions and deletions are common than searches. B^* trees are better when searching is the common tree operations. According to our statistics B trees need the same height, splits and nodes as B^* trees in some cases. We also found some cases in which B trees need more of height, splits and nodes than B^* trees. But B trees always need less comparisons than B^* trees. B^+ trees need more height, splits, nodes and comparisons than B and B^* trees. But B^+ trees are acceptable and good in applications in which both direct and sequential processing is required. This is because of using both a tree and a sequence set in B^+ trees.

DISTRIBUTED DATABASE

4.1 Introduction

In recent years, distributed databases have become an important area of information processing, and it is easy to foresee that their importance will rapidly grow. There are both organizational and technological reasons for this trend: distributed databases eliminate many of the shortcomings of centralized databases and fit more naturally in the decentralized structures of many organizations.

4.2 Distributed Database

A distributed database is a collection of data, which belong logically to the same system but are spread over the sites of a computer network.

For example, consider a bank that has three branches at different locations. At each branch, a computer controls the teller terminals of the branch and the account database of the branch. Each computer with its local account database at one branch constitutes one site of the distributed database, computers are connected by a communication network. During normal operations the applications which are requested from the terminals of a branch need only to access the database of that branch. These applications are completely executed by the computer of the branch where they are issued, and will therefore be called local applications. An example of a local application is a debit or a credit application performed on an account stored at the same branch at which the application is requested. Some applications are called global applications or distributed applications. A typical global application is a transfer of funds from an account of one branch to an account of another branch. This application requires updating the databases at two different branches.

Therefore, a distributed database is a collection of data distributed over different computers of a computer network. Each site of a network has autonomous processing capability

and can perform local applications. Each site also participates in the execution of at least one global application, which requires accessing data at several sites using a communication subsystem [7]. Figure 4.1 shows a typical Distributed Database.

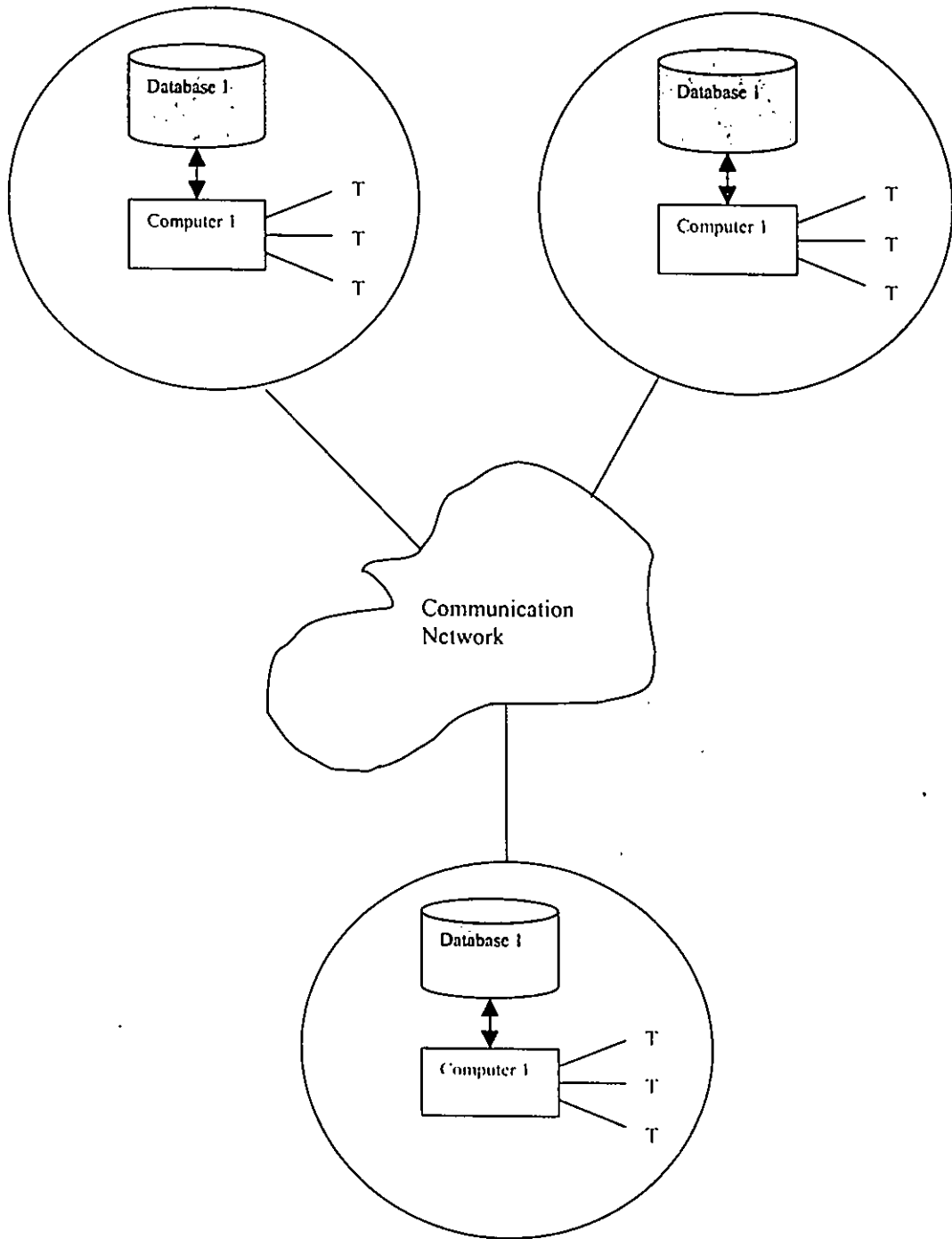


Figure 4.1: A typical example of Distributed Database

4.3 Distributed Database Management Systems (DDBMS)

A distributed database management system supports the creation and maintenance of distributed databases.

Several commercially available distributed systems were developed by the vendors of centralized database management systems[2][14][23]. They contain additional components, which extend the capabilities of centralized database management systems. They contain additional components, which extend the capabilities of centralized DBMSs by supporting communication and cooperation between several instances of DBMSs, which are installed at different sites of a computer network. The software components which are typically necessary for building a distributed database in this case are:

1. The database management component (DB)
2. The data communication component (DC)
3. The data dictionary (DD), which is extended to represent information about the distribution of data in the network.
4. The distributed database component (DDB)

These components are connected as shown in Figure 4.2 for a two-site network.

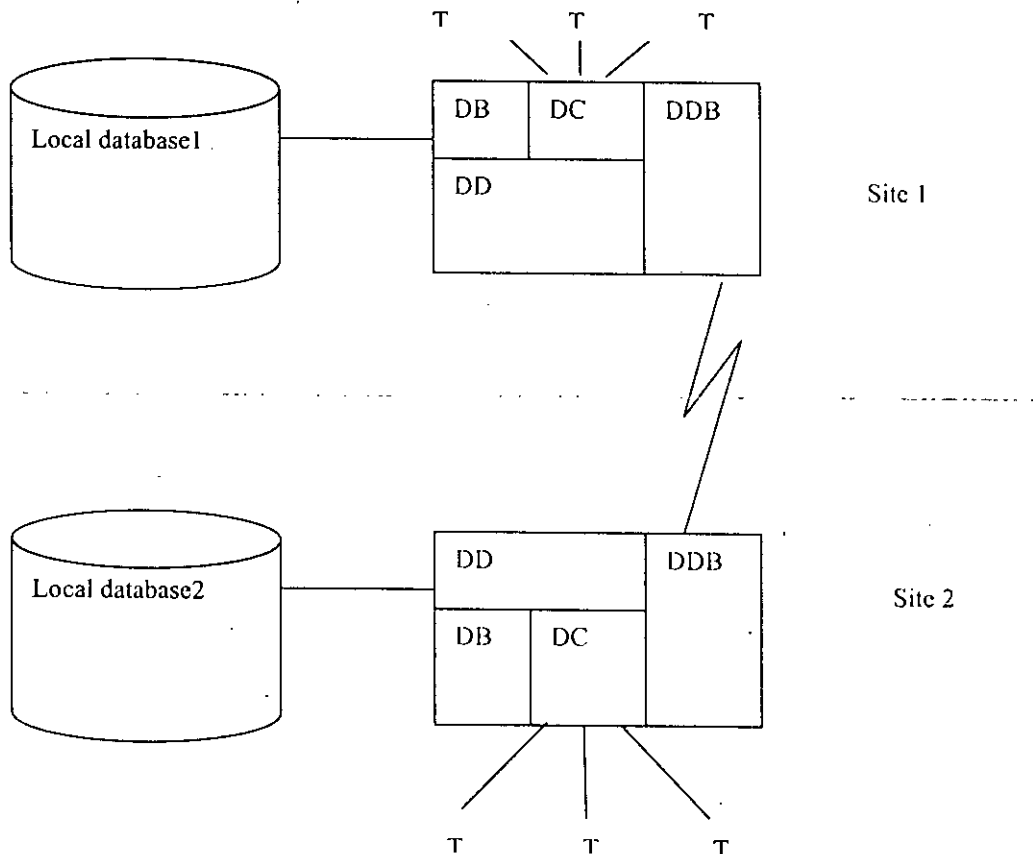


Figure 4.2: Software Components of Distributed Database Management Systems

4.4 Reference Architecture for Distributed Databases

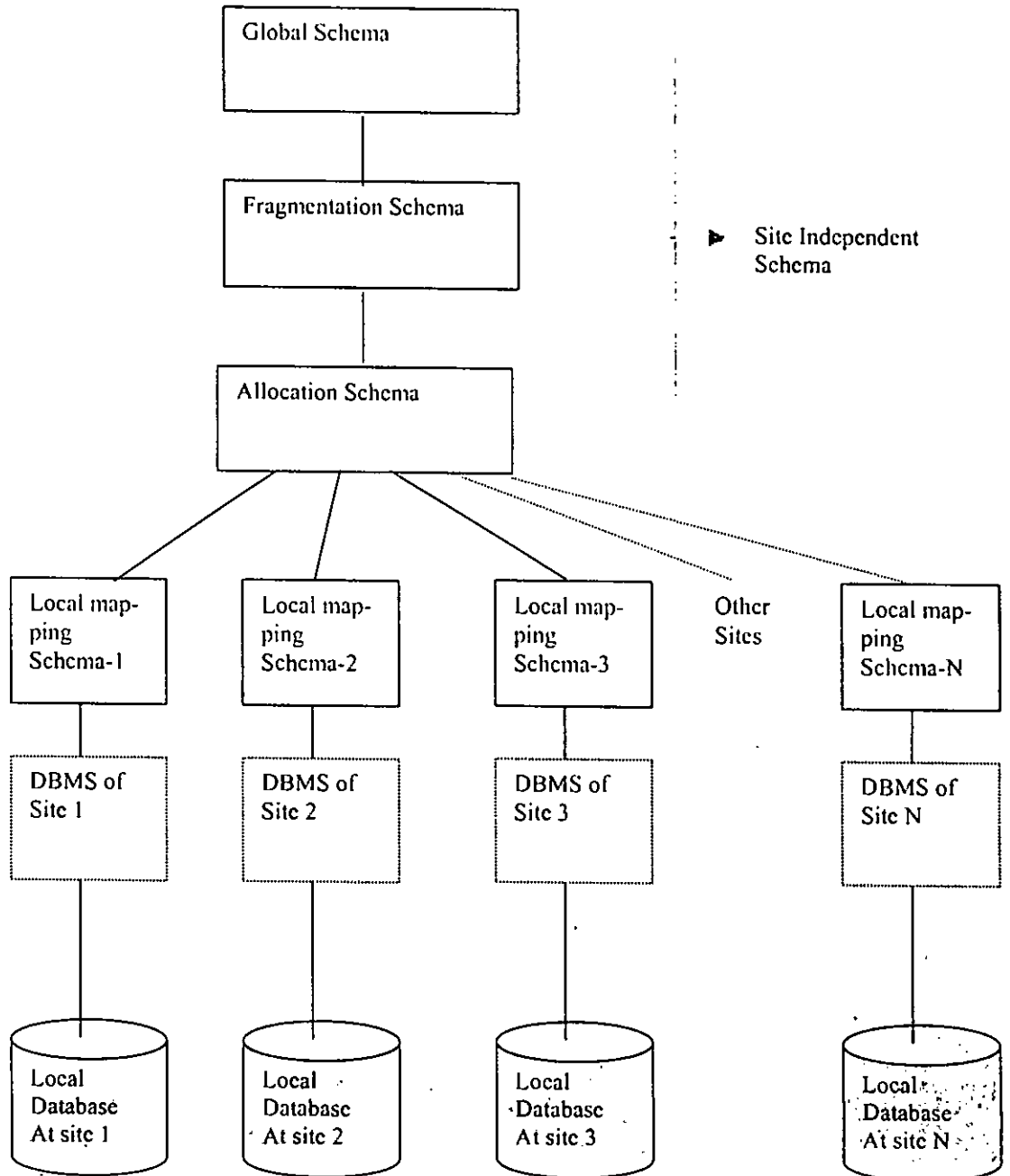


Figure 4.3: Reference Architecture of Distributed Database

Figure 4.3 shows reference architecture for a distributed database. This reference architecture is not explicitly implemented in all distributed databases; however, its levels are conceptually relevant in order to understand the organization of any distributed database.

At the top level of Figure is the global schema. The global schema defines all the data, which is contained in the distributed database as if the database were not distributed at all. For this reason, the global schema can be defined exactly in the same way as in a non-distributed database. However, the data model, which is used for the definition of a global schema, should be convenient for the definition of the mapping to the other levels of the distributed database. We will use the relational model for this purpose. Using this model, the global schema consists of the definition of a set of global relations.

Each global relation can be split into several non-overlapping portions, which are called fragments. There are several ways in which to perform the splitting operation. The mapping between global relations and fragments is defined in the fragmentation schema. This mapping is one to many; i.e., several fragments are corresponds to one fragment. Fragments are indicated by a global relation name with an index (fragment index); for example, R_i indicates the i th fragment of global relation R .

94603
Fragments are logical portions of global relations, which are physically located at one or several sites of the network. The allocation schema defines at which site(s) a fragment is located. Note that the type of mapping defined in the allocation schema determines whether the distributed database is redundant or non-redundant; in the former case the mapping is one to many, while in the latter case the mapping is one to one. All the fragments, which correspond to the same global relation R and are located at the same site j constitute the physical image of global relation R at site j . There is therefore a one to one mapping between a physical image and a pair \langle global relation, site \rangle ; physical images can be indicated by a global relation name and a site index. To distinguish them from fragments, we will use a superscript; for example, R^j indicates the physical image of the global relation R at site j . In figure 4.4 , a global relation R is split into four fragments are allocated R_1, R_2, R_3 & R_4 . These four fragments are allocated redundantly at the three sites of a computer network, thus building three physical images R^1, R^2, R^3 .

We will refer to a copy of a fragment at a given site, and denote it using the global relation name and two indexes. For example, the notation R^3_2 indicates the copy of fragment R_2 that is located at site 3. Finally, note that two physical images can be identical. In this case we will say that a physical image is a copy of another physical image. In figure 4.4, R^1 is a copy of R^2 .

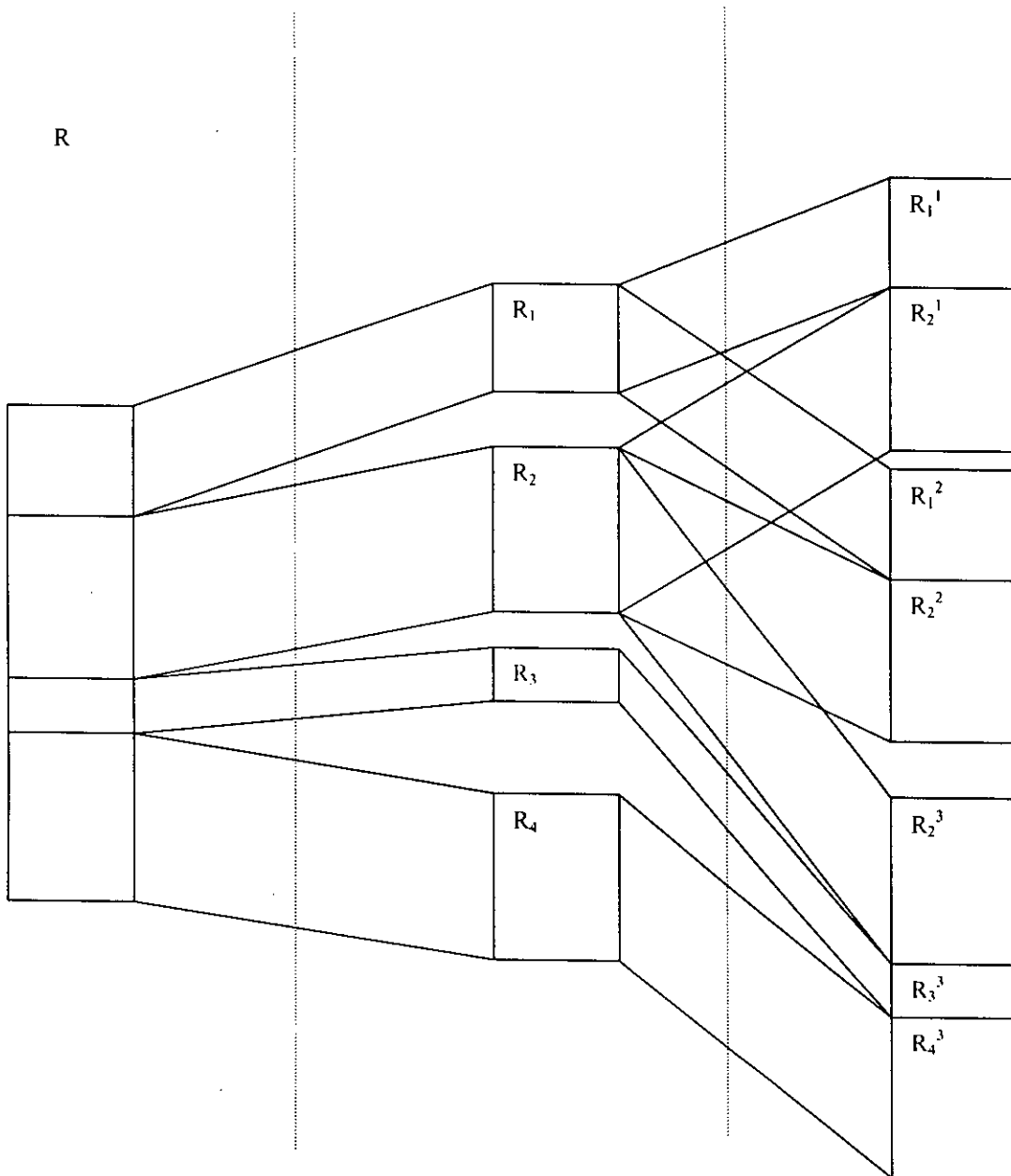


Figure 4.4 : Fragments and physical images for a global relation

Let us now go back to the reference architecture of figure 4.3. We have already described the relationships between the objects at the three top levels of this architecture. These three levels are independent; therefore they do not depend on the data model of the local DBMSs. At a lower level, it is necessary to map the physical images to the objects, which are manipulated by the local DBMSs. This mapping is called a local mapping schema and depends on the type of local DBMS; therefore in a heterogeneous system we have different types of local mappings at different sites.

This architecture provides a very general conceptual framework for understanding distributed databases. The three most important objectives, which motivate the features of this architecture, are the separation of data fragmentation and allocation, the control of redundancy, and the independence from local DBMSs.

4.5 Types of data fragmentation

The decomposition of global relations into fragments can be performed by applying two different types of fragmentation: horizontal fragmentation and vertical fragmentation.

In all types of fragmentation, a fragment can be defined by an expression in a relational language, which takes global relations as operands and produces the fragment as result. For example, if a global relation contains data about employees, a fragment which contains only data about employees who work at department D1 can be obviously defined by a selection operation on the global relation. There are, however, some rules, which must be followed when defining fragments:

Completeness condition: All the data of the global relation must be mapped into the fragments; i. e., it must not happen that a data item, which belongs to a global relation, does not belong to any fragment.

Reconstruction condition: It must always be to reconstruct each global relation from its fragments. The necessity of this condition is obvious: in fact, only fragments are stored in the distributed database, and global relation has to be built through this reconstruction operation if necessary.

Disjointness condition: It is convenient that fragments be disjoint, so that the replication of data can be controlled explicitly at the allocation level. However, this condition is useful mainly with horizontal fragmentation, while for vertical fragmentation we will sometimes allow this condition to be violated.

4.5.1 Horizontal Fragmentation

Horizontal fragmentation consists of partitioning the tuples of a global relation into subsets; this is clearly useful in distributed databases, where each subset can contain data,

which have common geographical properties. It can be defined by expressing each fragment as a selection operation on the global relation. For example, let a global relation be

$$\text{SUPPLIER}(\text{SNUM}, \text{NAME}, \text{CITY})$$

Then the horizontal fragmentation can be defined in the following way:

$$\text{SUPPLIER1} = \text{SL}_{\text{CITY} = \text{"SF"}} \text{SUPPLIER}$$
$$\text{SUPPLIER2} = \text{SL}_{\text{CITY} = \text{"LA"}} \text{SUPPLIER}$$

The above fragmentation satisfies the completeness condition if “SF” and “LA” are the only possible values of the CITY attribute; otherwise we would not know to which fragment the tuples with other CITY values belong.

The reconstruction condition is easily verified, because it is always possible to reconstruct the SUPPLIER global relation through the following operation:

$$\text{SUPPLIER} = \text{SUPPLIER1} \cup \text{SUPPLIER2}$$

The disjointness condition is clearly verified.

We will call the predicate, which is used in the selection operation, which defines a fragment, its qualification. For instance, in the above example the qualifications are:

$$q1 : \text{CITY} = \text{"SF"}$$
$$q2 : \text{CITY} = \text{"LA"}$$

We can generalize from the above example that in order to satisfy the completeness condition, the set of qualifications of all fragments must be complete, at least with respect to the set of allowed values. The reconstruction condition is always satisfied through the union operation, and the disjointness condition requires that qualifications be mutually exclusive.

4.5.2 Vertical Fragmentation

The vertical fragmentation of a global relation is the subdivision of its attributes into groups; fragments are obtained by projecting the global relation over each group. This can

be useful in distributed databases where each group of attributes can contain data, which have common geographical properties. The fragmentation is correct if each attribute is mapped into at least one attribute of the fragments; moreover, it must be possible to reconstruct the original relation by joining the fragments together. Consider, for example, a global relation

$$\text{EMP (EMPNUM, NAME, SAL, TAX, MGRNUM, DEPTNUM)}$$

A vertical fragmentation of this relation can be defined as

$$\text{EMP1} = \text{PJ}_{\text{EMPNUM, NAME, MGRNUM, DEPTNUM}} \text{EMP}$$
$$\text{EMP2} = \text{PJ}_{\text{EMPNUM, SAL, TAX}} \text{EMP}$$

This fragmentation could, for instance, reflect an organization in which salaries and taxes are managed separately. The reconstruction of relation EMP can be obtained as

$$\text{EMP} = \text{EMP1} \text{ JN}_{\text{EMPNUM}=\text{EMPNUM}} \text{EMP2}$$

4.6 Finding records in a Distributed Database

At present distributed databases are inefficient in locating records since it is not using any global index structure. For example, if we have a book data file in a distributed database, the single book data file should be fragmented into several data files and these fragments should be allocated in different sites of the distributed database. Name of the original non-fragmented data file and the names of the fragments will be stored in the fragmentation schema. And the information regarding the allocation of fragments to the sites will be stored in the allocation schema. When a query is searching a book by a particular author it will refer the non-fragmented data file. The fragmentation schema will tell the number and the names of the fragments of the data file. And the allocation schema will tell about the sites where to get the said fragments. Using these information Distributed Database Management Systems collect the pertaining fragments from different sites and reconstruct the data file as if it were not fragmented. After reconstruction every record will be read one after and the author name will be checked against a particular author to identify the

desired record. Reconstruction may not produce ordered file even if the fragments are ordered themselves. Query optimization techniques and efficient access strategy may exclude one or more fragments, still, there will be fragments that are used in the reconstruction but don't contain the record being searched. Alternatively, Distributed Database Management Systems can search the record in every fragments one after another. In both of these cases lot of extra works need to be done to find a record. To eliminate these extra works during searching a record an efficient index structure can be used.

PROPOSED DISTRIBUTED INDEX

5.1 Introduction

The reason for providing indexes is to obtain fast and efficient access to data. Indexing is a data structure based technique for accessing records in a file. Indexes are often tree-structured because search operations are efficient and sequential processing is normally easy in this structure. A B-tree is a multi-way tree with bounds on the node size. The B-tree organization is particularly suitable because the search, insert, and delete operations are guaranteed efficient even in the worst case. Most of the centralized database management systems use B-tree or other types of index structures. But in distributed databases no index structure is used to obtain fast and efficient access to the data. Because it is very difficult to build and maintain such structures and because it is not convenient to navigate at a record level in distributed databases. Therefore, efficient access is the main problem in distributed databases.

5.2 Distributed Index Architecture

To provide efficient access to the data we proposed a distributed index concept. Distributed index is also a data structure based index comprising of two types of index structures. One is Global Index (GI) and the other is Local Index (LI). Figure 5.1 shows the proposed distributed structure:

GI is created and maintained by distributed database component (DDB) of distributed database management systems (DDBMS). LI is created and maintained by local database management component (DB) of DDBMS. Our study shows that B-tree is most suitable index structure even in the worst case. For this reason we preferred B-tree for implementing both GI and LI of distributed index. LI has been implemented by B-tree as it was stated by Bayer and McCreight [5]. All the algorithms like searching, delete and insert algorithms for LI were used without any modification. Moreover, we used two algorithms, one of these algorithms finds the minimum index value in the B-tree and the other algorithm finds the maximum index value in the B-tree. Algorithm 5.1 and 5.2 contain

the pseudo-code algorithm for finding above mentioned minimum and maximum values respectively. In LI we stored these two index values along with the tree address in a special record shown in the figure 5.2.

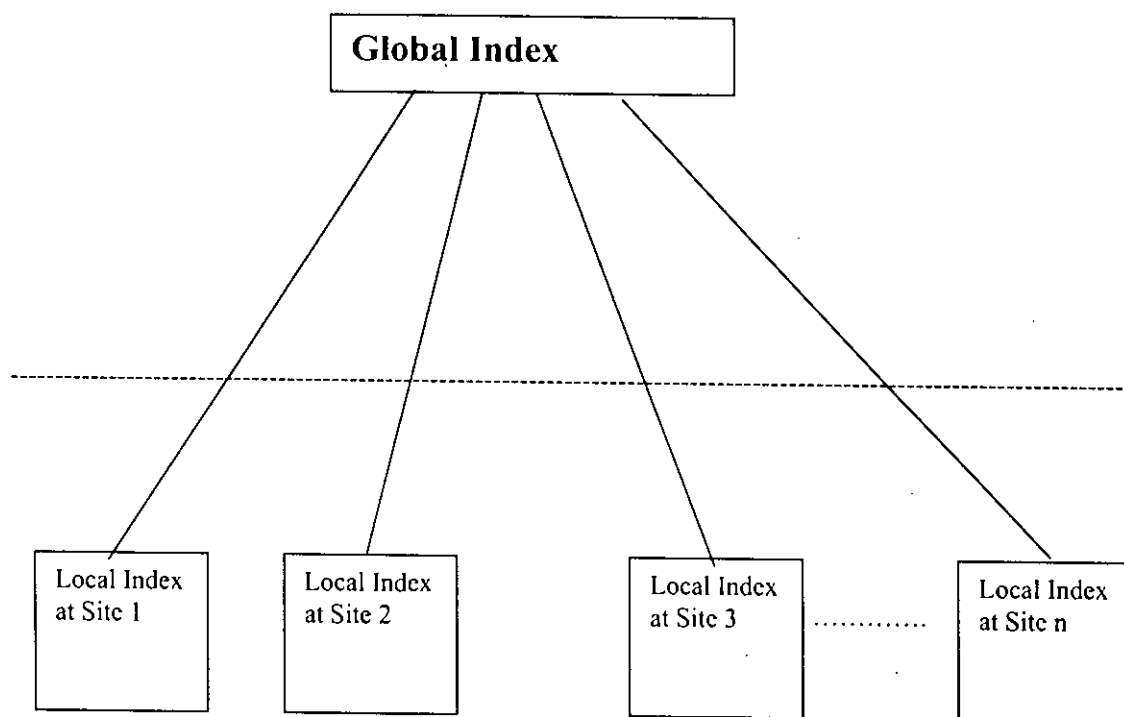


Figure 5.1: Architecture of Distributed Index

For every site there is a Local Index (LI), which has been created, updated and used independently. Like other local database management components LI enjoys autonomy in each site. There must be a single global index (GI) for a distributed index. GI is created, updated and used based on local indexes. All the local indexes are perfectly mapped with the global indexes. When a record is searched in a distributed database, GI used first to determine which LI needs to be used to find the data. After selecting the right LI it is used to access records in the corresponding site. In this way, distributed index ensures efficient access to the data in a distributed database. If a record is inserted, deleted or updated in a site special record in the site is modified properly. After any modification in LI above mentioned algorithms are always used to find the new minimum and maximum index values in the modified B-tree. The new minimum and maximum are checked against the stored (old) minimum and maximum values. If old and new minimum are not same or old and new maximum are not same, LI overwrites the new values in the special record, and

the local database management component passes this changed record to the distributed database management component to modify the GI accordingly.

Site address of B-tree (LI)	Minimum index value	Maximum index value
--------------------------------	---------------------	---------------------

Figure 5.2 Special record in LI

Find Minimum (treeRoot, min)

```

{
    currentNode = treeRoot
    leftChild =currentNode.P(0)
    min=currentNode.R(0)
    While (leftChild != Null)
    {
        currentNode = leftChild
        leftChild =currentNode.P(0)
        min= currentNode R(0)
    }
    return min
}

```

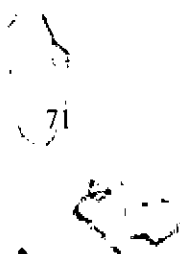
Algorithm 5.1: Find minimum index value in a B-tree

Find Maximum (treeRoot, max)

```

{
    currentNode = treeRoot
    norec = currentNode.rec
    rightChild = currentNode.P(norec)
    max = currentNode.R(norec-1)
    While (rightChild != Null)
    {
        currentNode = rightChild
        norec = currentNode.rec
    }
}

```



```

        max = currentNode R(norec-1)
    }
    return max
}

```

Algorithm 5.2: Find maximum index value in a B-tree

Proposed Global Index (GI) is almost a B tree in its structure and logic. Traditional B-tree definition is also adopted here. But the records in a GI node are different from that of a B tree. B tree holds the index value and the pointer/address of the concern record in the data file. But GI record holds the minimum and maximum index values of a local index (original B-tree in a site) and the pointer/address of that local index. In a B-tree, if a single index value points more than one records in the data file, the B-tree record keeps the index value and a pointer to a bucket that points all the concern records in the data file. Similarly, if a range of minimum and maximum index values refers more than one local index in different sites, the GI record keeps the index range and a pointer to a bucket that refers all the concern local indexes in different sites. We did not consider this complex situation in this thesis work. We have developed and tested algorithms for the simple case where an index range always refers a single local index in a single site. Search, insert and delete algorithms for GI is different from those of B-tree. We have proposed these algorithms and tested their effectiveness.

5.3 Distributed Index searching (GI searching)

When searching for a record with a given key value we start searching in global index first to find out the right local index. We start searching the global index by examining the root node. We search the node for the record whose minimum index value and maximum index value represents the range in which the search key falls. When we find such a record we get the pointer/location of the local index in the record. Using that pointer/location we start searching in the local index for a record with the search key. The result of this search in the local index is reported to the global index.

If the record with the right range is not found in the global index, comparisons of minimum index values and maximum index values in the node with the search key will find out the pointer to the sub-tree that may contain the record. We read the node pointed to and repeat the operations. If the pointer is null we report that the record we are searching for is not present in the distributed database. Algorithm 5.3 contains the pseudo-code algorithm for searching record in a GI.

Search GI (GIroot , searchKey)

```

{   currentNode = GIroot
    found = False

    while (currentNode != null)
    {
        norec = currentNode.rec
        for ( i=0; i<norec; i++)
        {
            currentRecord =      currentNode. R(i)
            if (searchKey < currentRecord.min
                currentNode =currentNode.P(i)
            elseif (currentRecord.min≤searchKey ≤currentRecord.max)
            {   Search in the LI pointed by currentRecord.P
                search LI (LI, searchKey, found)
            }
            elseif (i= (norec - 1))
                currentNode=currentNode.P(norec)
        }

        if (currentNode == null)
            found = False
    }

    if (found)

```

```

    return (record)
}

```

Algorithm 5.3: Algorithm for searching in Global Index of a Distributed Database

5.4 GI insertion

Insertion algorithm in GI is almost same as that of B-tree. Insertion algorithm of B-tree compares the key value of the record needs to be inserted with the key values of the records in different node. Insertion algorithm of GI does the same thing by using either the minimum values or maximum values for comparison. This would certainly return the correct insertion point in a terminal node. Only constraint is that if the algorithm is using the minimum values for comparison it has to use the minimum values always. In this thesis work we use minimum values. The algorithm solves the overflow problem in a node in the same way as it is solved in B-tree insertion algorithm.

InsertGI (GIroot, inRec)

```

{
    found = false
    currentNode = GIroot
    repeat
    { N=currentNode.norec
      currentRecord=currentNode.R(0)
      case
        inRec.min = currentRecord.min: found = true
        inRec.min < currentRecord.min: P = currentNode.P(0)
        inrec.min > currentNode.R(N).min: P = currentNode.P(N)
        otherwise
          P = currentNode.P(i - 1) for some i where
            CurrentNode.R(i - 1).min < inRec.min <currentNode.R(i).min
      endcase
      If P not null then push onto stack address of currentNode
      CurrentNode = P
    }
}

```

```

    } until (found) or (P = null)
if (found) then the record is already in the tree
else
    {
        P = null
        Finished = false
        Repeat
            {
                if currentNode is not full then put inRec and P in currentNode
                    Finished = true
                else copy currentNode to TOOBIG
                    insert inRec and P into TOOBIG
                    inRec = center record of TOOBIG
                    currentNode = 1st half of TOOBIG
                    Get space for newNode, assign address to P
                    newNode = 2nd half of TOOBIG
                    If stack not empty then pop to stack and read node pointed to
                    else
                        get space for new node
                        newNode = pointer to oldRoot, inrec and P
                        Finished = True
            }
        until Finished
    }
}

```

Algorithm 5.4: Algorithm for inserting records in Global Index of a Distributed Database

5.5 GI deletion

Deletion algorithm in GI is almost same as that of B-tree. Deletion algorithm of B-tree like insertion algorithm compares the key value of the records needs to be deleted with the key values of the records in different nodes to find out the right record and pointer to delete. Deletion algorithm of GI does the same thing by comparing either the minimum values or maximum values. In this thesis we compare minimum values. Like B-tree deletion algorithm if the record we need to delete is not in a terminal node this algorithm finds its successor in a terminal node and swaps these two records and then deletes the record and the pointer from the terminal node. Underflow problem in anode is solved by this algorithm in the same way as it is solved in a B-tree deletion algorithm. Algorithm 5.5 contains the pseudo-code algorithm for deleting from a GI.

```
DeleteGI (GIroot, outRec)
{
found = false
currentNode = GIroot
repeat
    {
    N = currentNode.norec
    CurrentRecord = currentNode.R(O)
    case
        outRec.min = currentRccord. min: found = true
        outRec.min < currentRecord. min: P = currentNode.P(0)
        outRec.min > currentNode.R(N). min: P = currentNode.P(N)
    otherwise
        P = currentNode.P(i - 1) for some i where
            CurrentNode.R(i - 1).min < outRec.min <currentNode.R(i).min
    endcase
    if P not null then push onto stack the address of currentNode
        CurrentNode = P
    } until (found) or P is not null
if (found)
```



```

{ if outRec is not in terminal node
    {then search for successor record of outRec at terminal level (stacking
        node addresses)
    copy successor over outRec
    terminal node successor now becomes the outRec
    }
Finished = false
repeat
{remove outRec and pointer P
if currentNode is not root or is not too small then finished = True
elseif redistribution possible then
    { copy "best" A-sibling, intermediate parent-record, and currentNode into
        TWOBNODE
    copy records and pointer from TWOBNODE to "best" A-sibling, parent,
        and currentNode so A-sibling and currentNode are roughly equalize
    Finished = True
    }
else
    { choose best A-sibling to concatenate with put in the leftmost of the
        currentNode and A-sibling the contents of both nodes and the
        intermediate record from the parent discard rightmost of the two nodes
        intermediate record in parent now becomes outRec.
    }
until finished
if no records in root
    then { new root is the node pointed to by the current root
        discard oldroot}
}
}

```

Algorithm 5.5: Algorithm for deleting record from Global Index of a Distributed Database

EXPERIMENTAL RESULT OF PROPOSED DISTRIBUTED INDEX

6.1 Introduction

We discussed our proposed distributed index structure for distributed database in chapter 5. Our proposed structure will be effective on simple non-overlapping distributed databases only. To show the effectiveness and correctness of our proposed distributed index structure we have developed a simulation program. The results from the program are shown in this chapter.

6.2 Simulation Program

Developed simulation program can take a simple non-overlapping distributed relation as input. For each site it creates a local index of the records in that site. Based on these local indices from all sites the program creates the global index. Using the global and local indices the program can search, insert, and delete records in the distributed relation efficiently.

6.3 Results

We considered a simple non-overlapping distributed bank database, which has a global relation account as follows:

ACCOUNT (ACCNUM, ACCNAME, CUSTNUM, OPENDATE, BALANCE)

Where the attribute ACCNUM is any valid account number, the attribute ACCNAME is any valid account name, the attribute CUSTNUM is any valid customer number, the attribute OPENDATE is the opening date of the account and the attribute BALANCE is the balance amount of the account. The account table is necessary to be indexed by ACCNUM. Let the bank has several branches and the distributed site is maintained in every branch.

ACCOUNT schema has been fragmented into 10 fragments as follows:

$ACCOUNT_1 = SL_{001 \leq ACCNUM \leq 100} ACCOUNT$
 $ACCOUNT_2 = SL_{101 \leq ACCNUM \leq 200} ACCOUNT$
 $ACCOUNT_3 = SL_{201 \leq ACCNUM \leq 300} ACCOUNT$
 $ACCOUNT_4 = SL_{301 \leq ACCNUM \leq 400} ACCOUNT$
 $ACCOUNT_5 = SL_{401 \leq ACCNUM \leq 500} ACCOUNT$
 $ACCOUNT_6 = SL_{501 \leq ACCNUM \leq 600} ACCOUNT$
 $ACCOUNT_7 = SL_{601 \leq ACCNUM \leq 700} ACCOUNT$
 $ACCOUNT_8 = SL_{701 \leq ACCNUM \leq 800} ACCOUNT$
 $ACCOUNT_9 = SL_{801 \leq ACCNUM \leq 900} ACCOUNT$
 $ACCOUNT_{10} = SL_{901 \leq ACCNUM \leq 1000} ACCOUNT$

Each of the above fragments is allocated in a different site, i.e., there are ten sites and site i holds the fragment $ACCOUNT_i$. Local Database management system at each site creates an index on each fragment by $ACCNUM$ using the Algorithm 2.2 and assuming the B Tree of order 3. We call these indexes Local Indexes (LI). Using Algorithm 5.1 and 5.2, local system creates the special record in each site. These special records and local indexes at site 1, site 2 and so on are shown in the Figure 6.1 to 6.10 respectively. Each local system passes the special record to the global database management component to construct the Global Index (GI). Construction of GI is nothing but inserting each special record in the proposed B Tree of order 3 using the Algorithm 5.4. Figure 6.11 shows GI step by step.

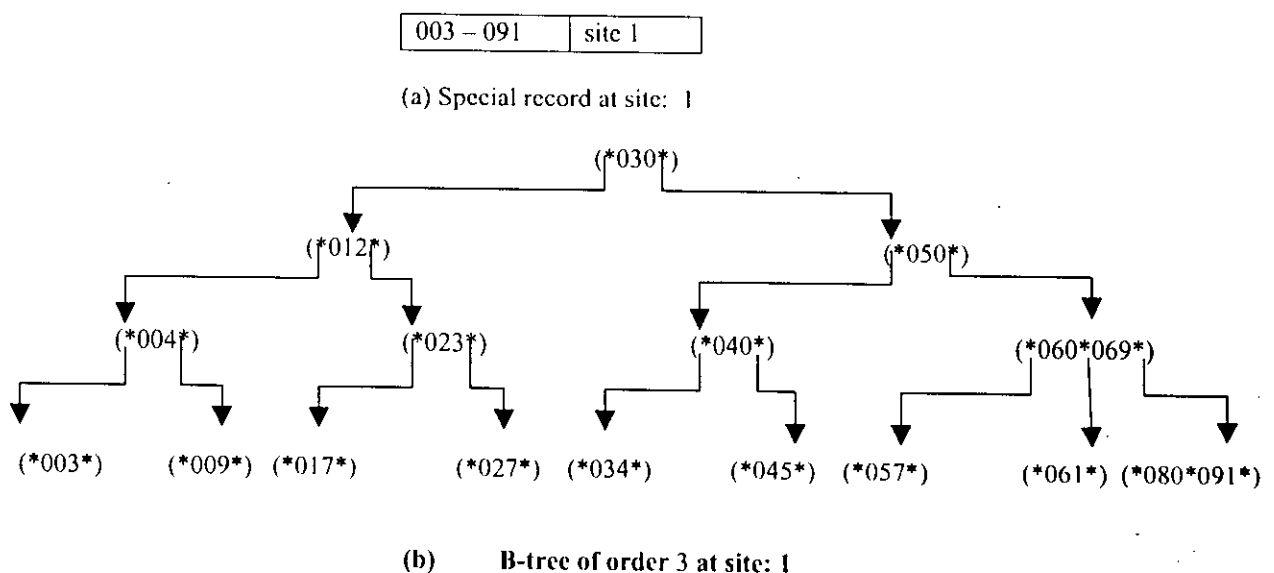
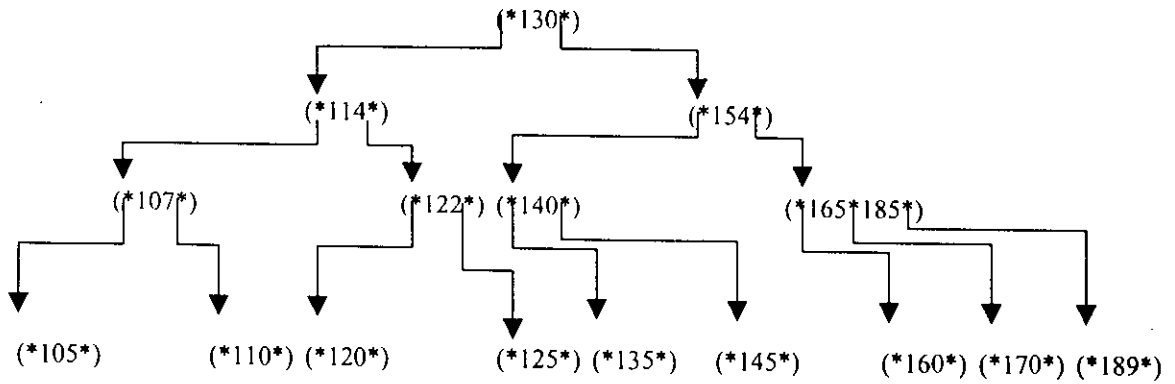


Figure 6.1: Local Index (local B Tree) at site 1

105 - 189	site 2
-----------	--------

(a) Special record at site: 2

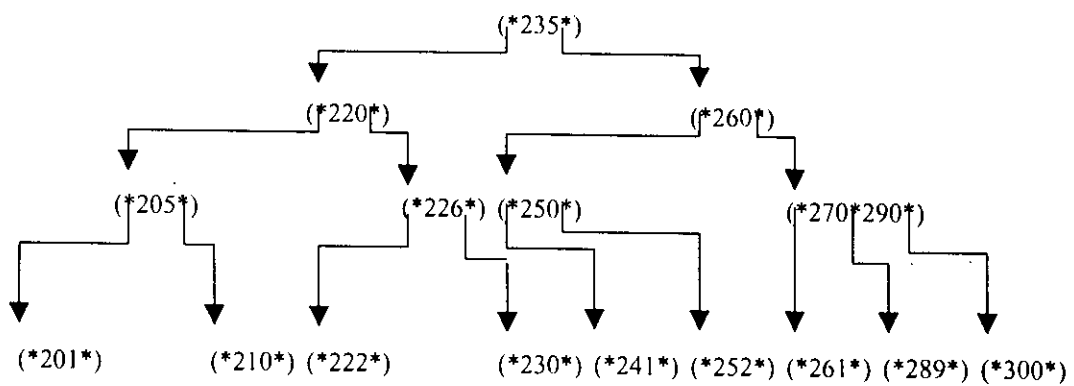


(b) B - Tree of order 3 at site: 2

Figure 6.2: Local Index (local B Tree) at site 2

201-300	Site3
---------	-------

(a) Special record at site: 3

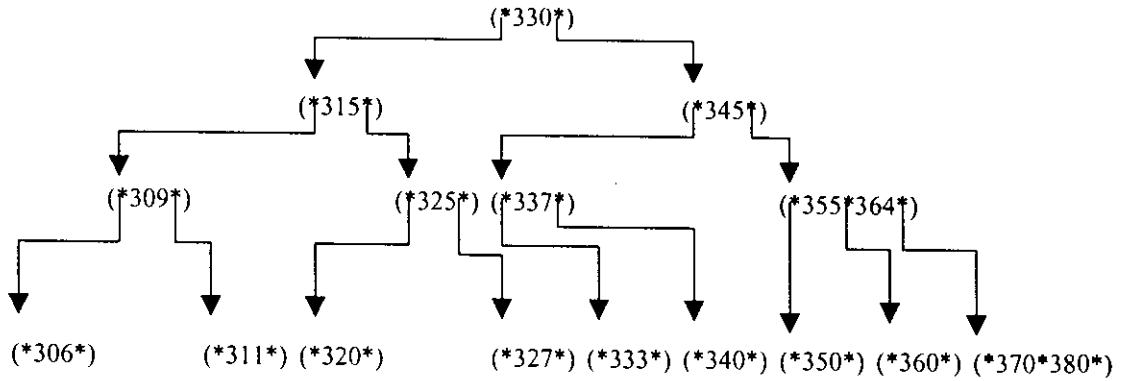


(b) B - Tree of order 3 at site: 3

Figure 6.3: Local Index (local B Tree) at site 3

306 - 380	site 4
-----------	--------

(a) Special record at site: 4

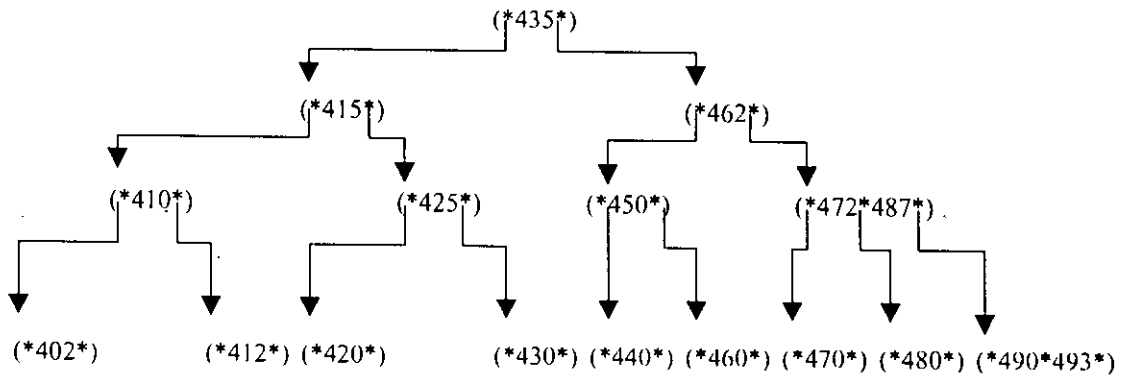


(b) B - Tree of order 3 at site: 4

Figure 6.4: Local Index (local B Tree) at site 4

402 - 493	site 5
-----------	--------

(a) Special record at site: 5

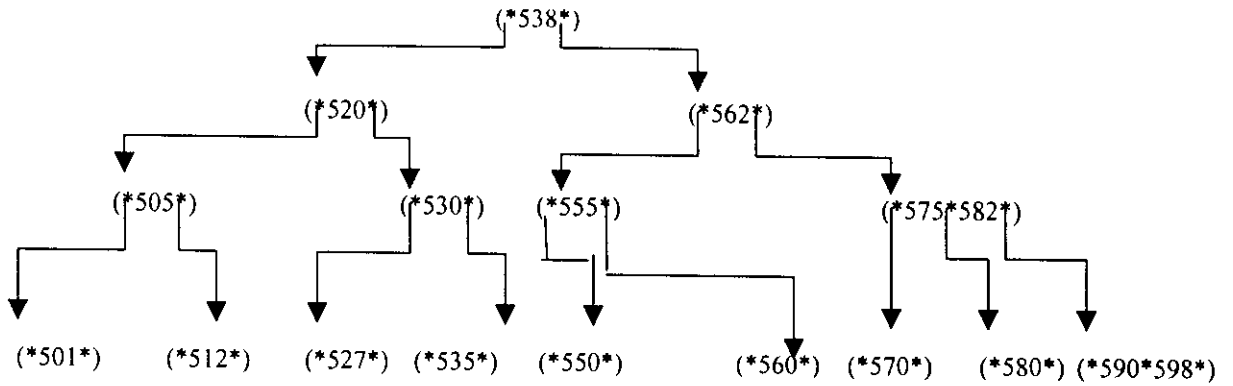


(a) B Tree of order 3 at site: 5

Figure 6.5: Local Index (local B Tree) at site 5

501 - 598	site 6
-----------	--------

(a) Special record at site: 6

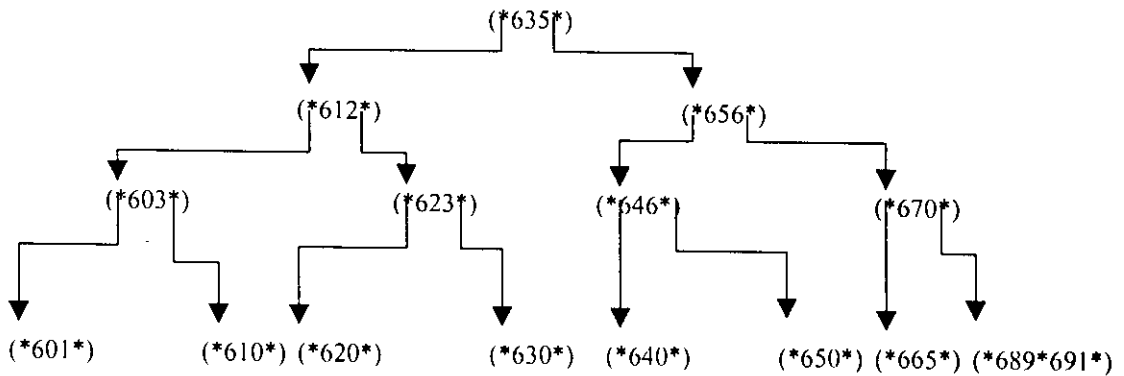


(a) B Tree of order 3 at site: 6

Figure 6.6: Local Index (local B Tree) at site 6

601 - 691	site 7
-----------	--------

(a) Special record at site: 7

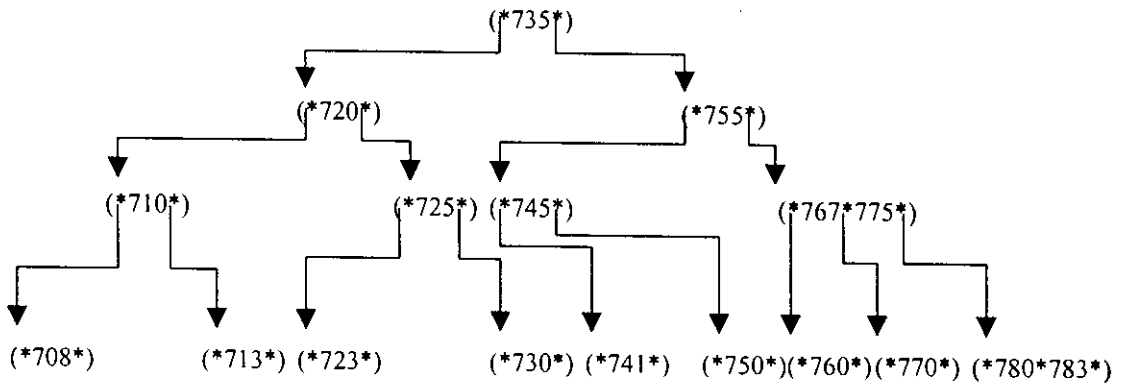


(a) B Tree of order 3 at site: 7

Figure 6.7: Local Index (local B Tree) at site 7

708 - 783	site 8
-----------	--------

(a) Special record at site: 8

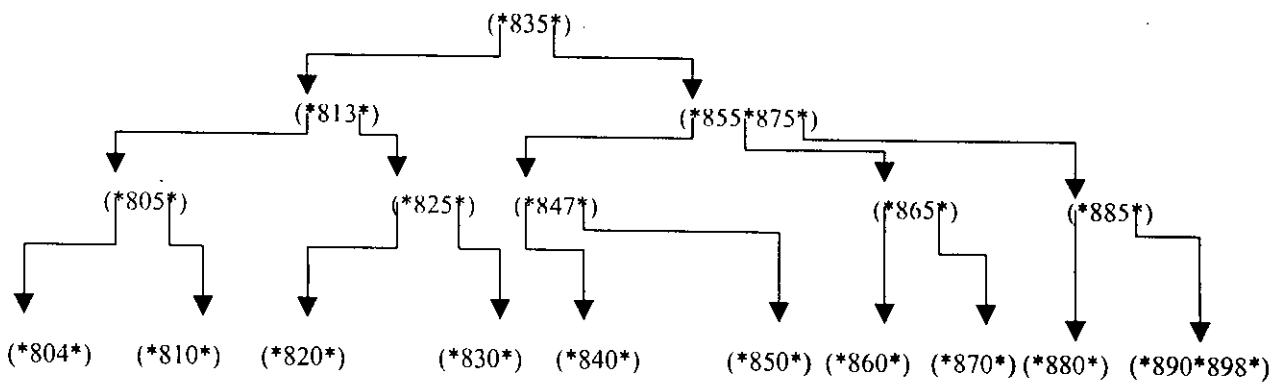


(a) B Tree of order 3 at site: 8

Figure 6.8: Local Index (local B Tree) at site 8

804 - 898	site 9
-----------	--------

(a) Special record at site: 9

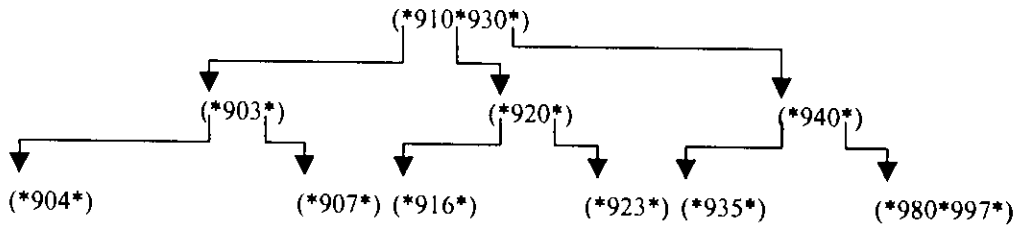


(a) B Tree of order 3 at site: 9

Figure 6.9: Local Index (local B Tree) at site 9

904 - 997	site 10
-----------	---------

(a) Special record at site: 10



(a) B Tree of order 3 at site: 10

Figure 6.10: Local Index (local B Tree) at site 10

{*(003 - 091, site 1)*}

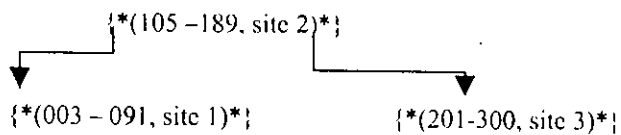
(a) Global Index after inserting the record coming from site 1

{*(003 - 091, site 1)**(105 - 189, site 2)*}

(b) Global Index after inserting the record coming from site 2

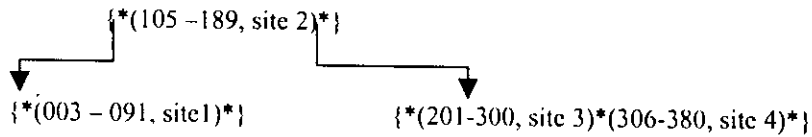
{*(003 - 091, site 1)**(105 - 189, site 2)**(201-300, site 3)*}
TOOBIG

(c) Global Index after inserting the record coming from site 3

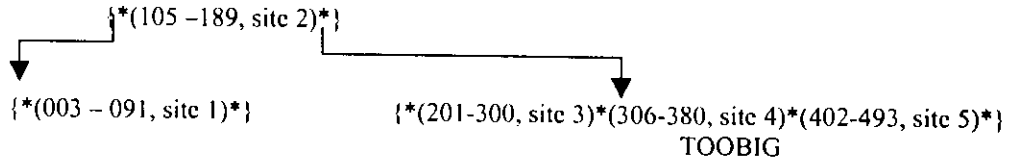


(d) Global Index after splitting the TOOBIG node

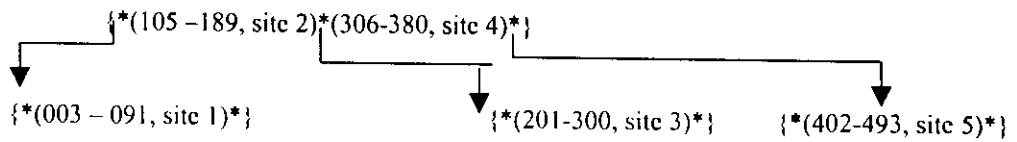
Figure 6.11(Part 1 of 4): Global Index after inserting records from site 1, 2, and 3



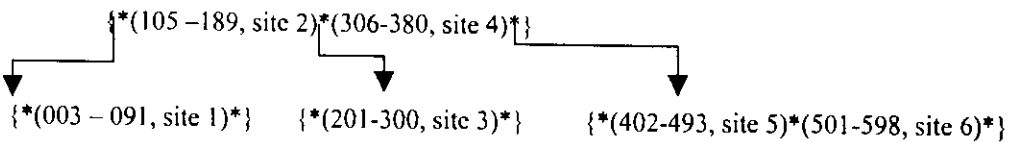
(e) Global Index after inserting the record coming from site 4



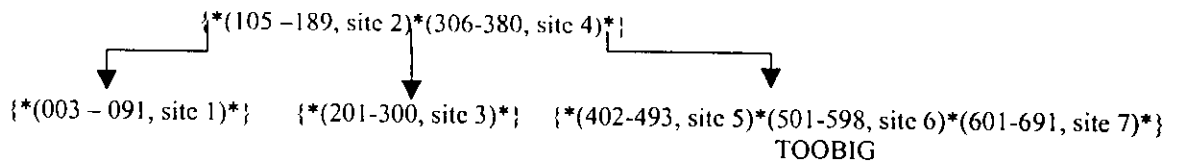
(f) Global Index after inserting the record coming from site 5



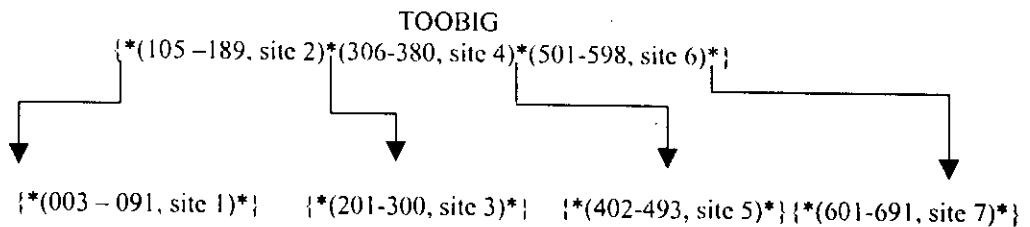
(g) Global Index after splitting the TOOBIG node



(h) Global Index after inserting the record coming from site 6

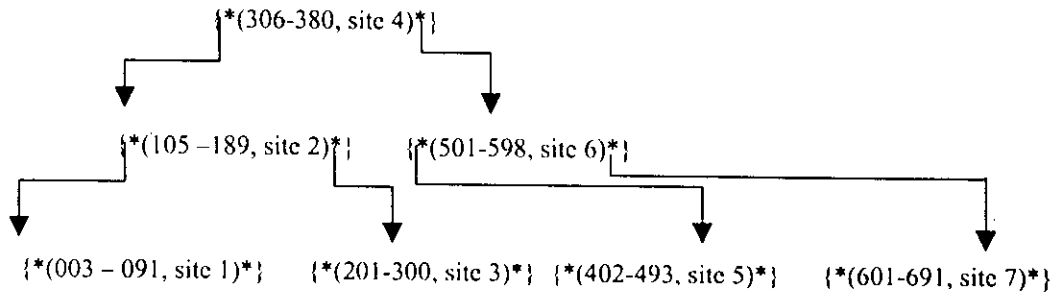


(i) Global Index after inserting the record coming from site 7

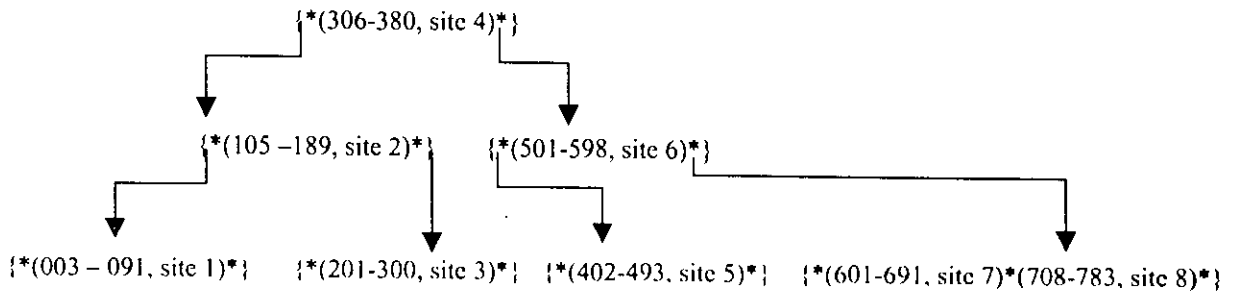


(j) Global Index after splitting TOOBIG node

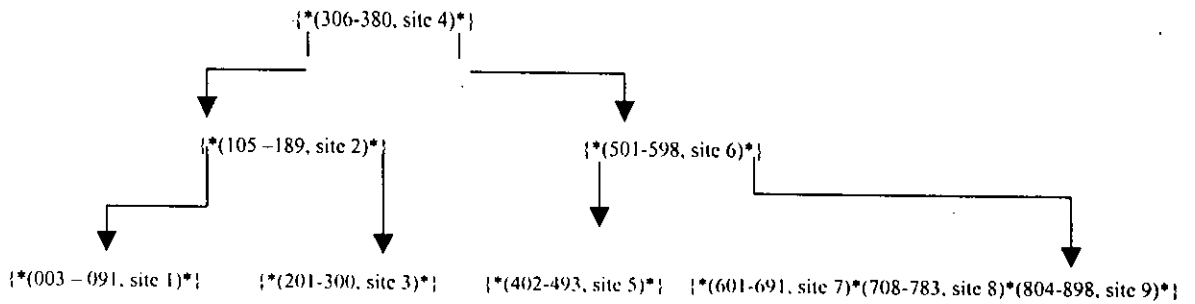
Figure 6.11(Part 2 of 4): Global Index after inserting records from site 4, 5, 6 and 7



(k) Global Index after splitting TOOBIG root

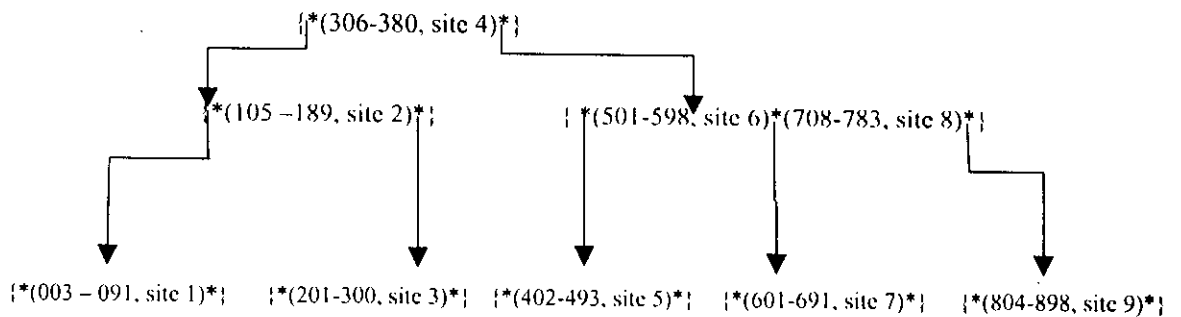


(l) Global Index after inserting the record coming from site 8



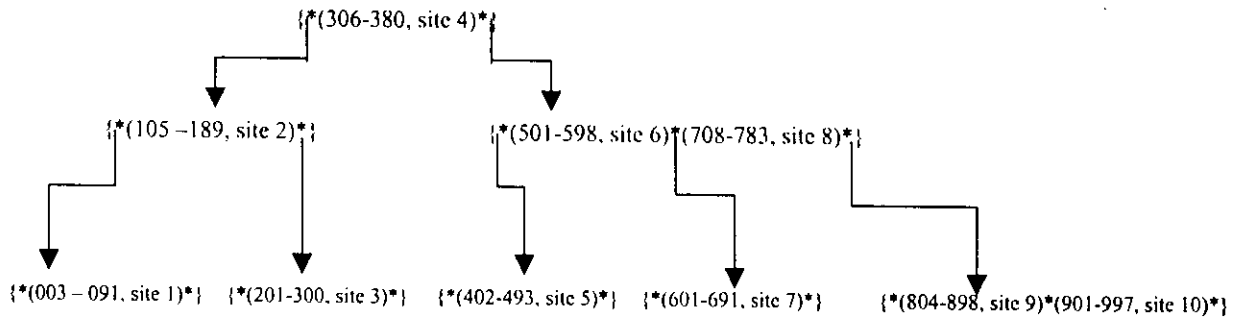
TOOBIG

(m) Global Index after inserting the record coming from site 9



(n) Global Index after splitting TOOBIG node

Figure 6.11(Part 3 of 4): Global Index after inserting records from site 8 and 9



(o) Global Index after inserting the record coming from site 10

Figure 6.11(Part 4 of 4): Global Index after inserting records from site 10

6.3.1 Searching account number 333

Every search would start from the root of the GI. The root node, in this case, has one record. The minimum and maximum values of this record are 306 and 380 respectively. 333 belong to this range. The search in GI by the Algorithm 5.3 will return the site 4 for further searching. Searching at site 4 by the Algorithm 2.1 will again start from the root of the LI at site 4. Algorithm 2.1 will eventually find the record with the account number 333 in the local index.

6.3.2 Searching account number 489

Algorithm 5.3 will not find any record in the root of GI in which 489 belongs. The root has only one record. The minimum and maximum values of this record are 306 and 380 respectively. 489 is greater than both 306 and 380. For this reason, Algorithm 5.3 will start checking the right child node of the visited record in the root. The first record in this node has the minimum value 501, which is larger than 489. So the search will proceed by checking the left child node. The only one record of this child node has the minimum and maximum values 402 and 493 respectively. 489 belongs to this range. Algorithm 5.3 will return site 5 for further searching. Searching at site 5 using the Algorithm 2.1 will not find any record for the account number 489. The answer of the search will be 'the account number 489 does not exist in the database'.

6.3.3 Searching account number 000

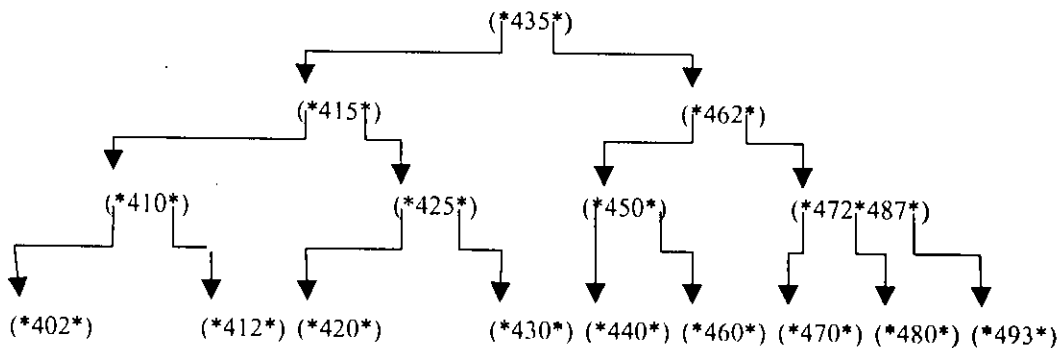
Algorithm 5.3 will start searching account number 000 at the root of the GI. The only record in the root has the minimum value 306, which is larger than 000. For this reason, the left child of the record will be checked. The only record in this child has the minimum value 105, which is again larger than 000. So the search will proceed by checking the left child of this record. The new child node has only one record with the minimum value 003, which is larger than 000. The search will try to check the descendant left child node. But there is no left child node in this case. As a result, the search will be end at this point without returning any site for further search. And the answer of the search is ' the account number 000 does not exist in the databasc'.

6.3.4 Deleting account number 490

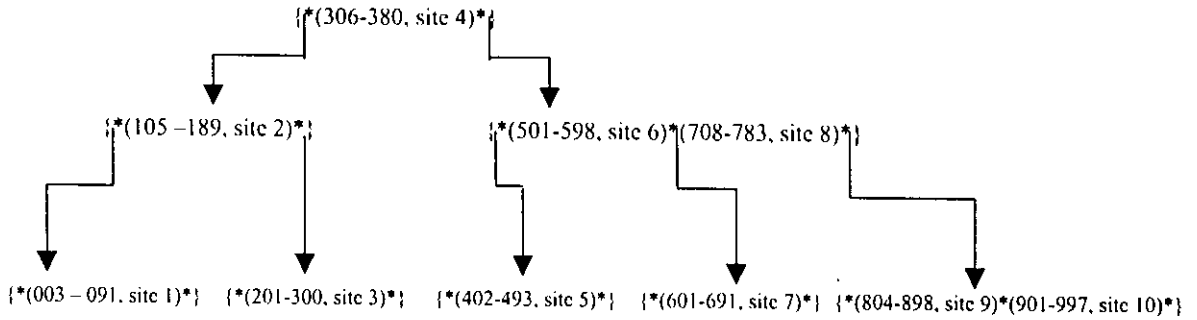
For deleting account number 490, it has to be searched first using the Algorithm 5.3 and Algorithm 2.1. The search will find the account at site 4. Using Algorithm 2.3 account number 490 will be deleted from the LI at site 4. Algorithm 5.1 and Algorithm 5.2 will give the new minimum and maximum values at site 4. These are 402 and 493 respectively. Since they are same as the old minimum and maximum values, the delete algorithm has nothing to do in GI. The delete will be simply complete at this stage. Figure 6.12 shows the LI and GI after deleting the account number 490.

402-493	Site 5
---------	--------

(a) Special record at site: 5 after deleting the account number 490



(b) Local Index (B Tree of order 3) at site: 5 after deleting the account number 490



(c) Global Index (Proposed B Tree of order 3) after deleting the account 490

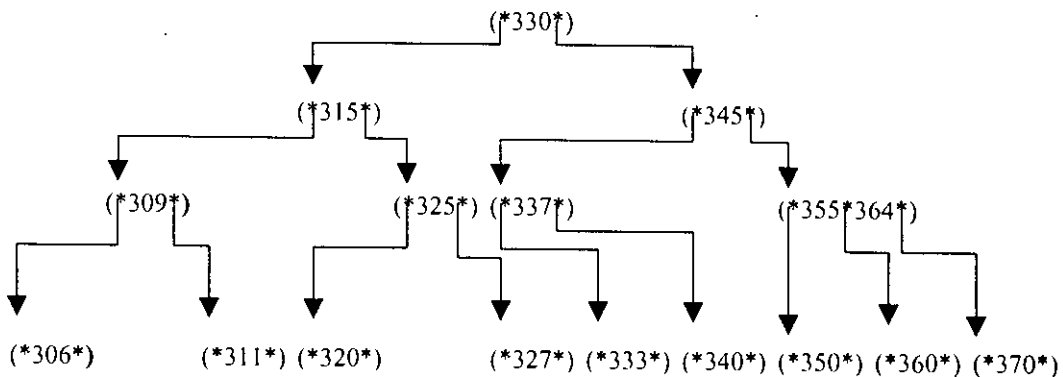
Figure 6.12: Local Index at site 5 and the Global Index after deleting the account 490

6.3.5 Deleting account number 380

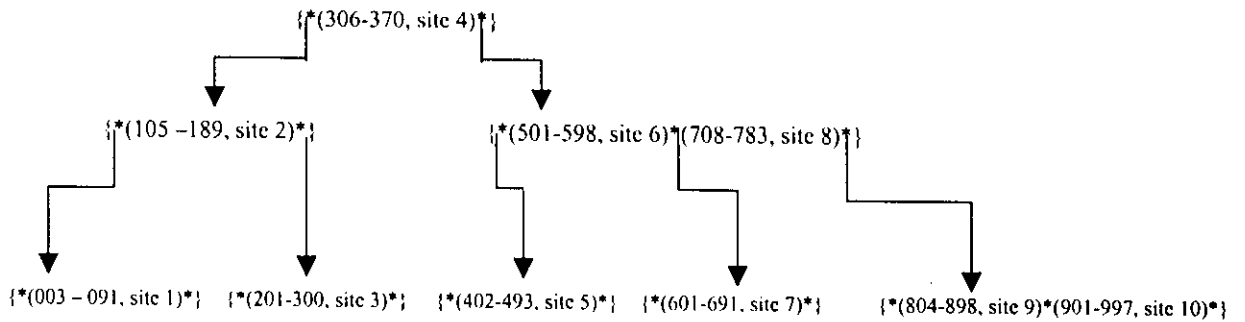
Algorithm 5.3 and 2.1 will find the account at site 4. Algorithm 2.3 will delete the account from the LI at site 4. Algorithm 5.1 and 5.2 will give the new range of the LI, which are 306 to 370. Before deleting the account (with the account number 380) the range was 306 to 380. Since the range has been changed it has to be reflected in the GI. Algorithm 5.3 will find a record in the GI with the range 306 to 380 and the site address as site 4. Using the Algorithm 5.5 this record will be deleted first. After that the Algorithm 5.4 will insert a record in the GI with the range 306 to 370 and the site address site 4. Figure 6.13 shows the LI and GI after deleting the account number 380.

306 - 370	site 4
-----------	--------

(a) Special record at site: 4 after deleting the account number 380



(b) Local Index (B - Tree of order 3) at site: 4 after deleting the account number 380



(c) Global Index (Proposed B Tree of order 3) after deleting 380

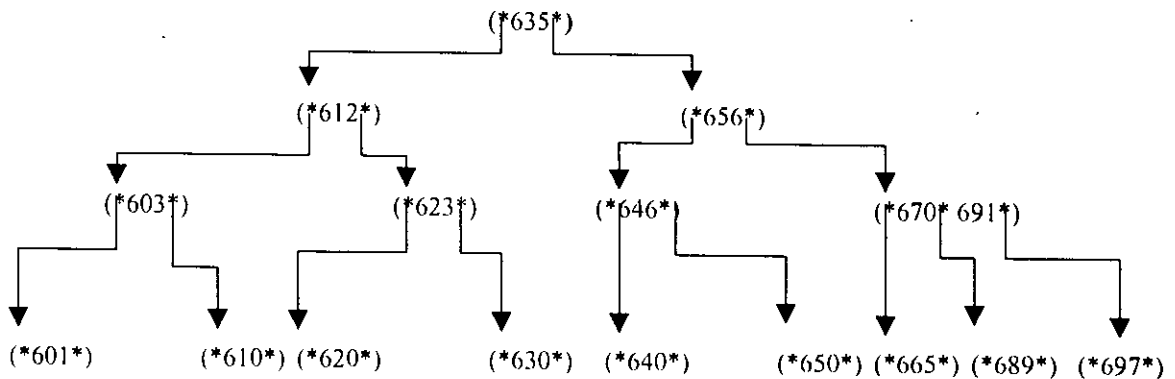
Figure 6.13: Local Index at site 4 and the Global Index after deleting the account 380

6.3.6 Inserting the account number 697 at site 7

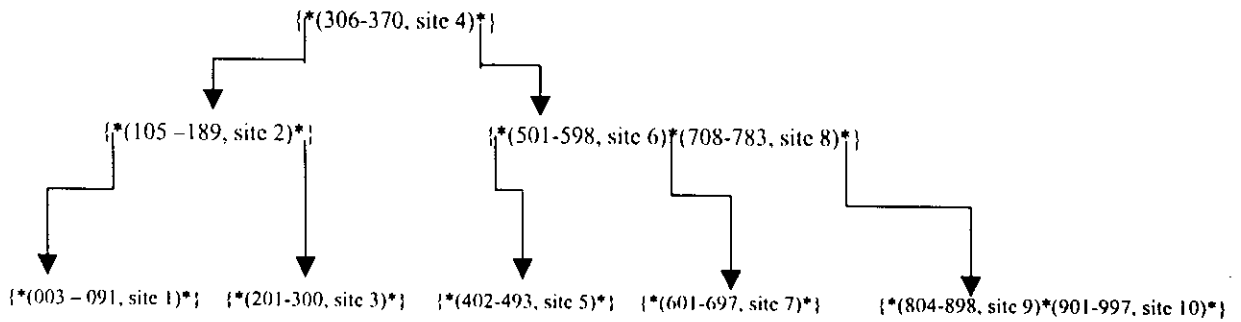
Algorithm 2.2 will insert the account number 697 at site 7. Algorithm 5.1 and 5.2 will give the new range of the LI at site 7. The range has been changed to 601- 697 from 601-691. The change needs to be reflected in the GI. Algorithm 5.3 will first find a record in the GI with old range and site address (601-691, site 7). Algorithm 5.5 will delete the record from the GI. Algorithm 5.4 will insert a new record in the GI with the new range and site address (601-697, site 7). Figure 6.14 shows the LI and GI after inserting the account number 697 at site 7.

601 - 697	site 7
-----------	--------

(a) Special record at site: 7 after inserting the account number 697



(b) Local Index (B Tree of order 3) at site: 7 after inserting the account number 697



(c) Global Index (Proposed B Tree of order 3) after inserting the account number 697

Figure 6.14: Local Index at site 7 and the Global Index after inserting the account number 697

6.3.7 GI after site 10 has been vanished

If all the records of a site, say site 10, have been deleted, the LI of the site is just a root node without any record and children. And the special record is simply an empty record. This situation needs to be reflected in the GI. For this, the Algorithm 5.3 will be used to find the record of site 10 in GI. And the Algorithm 5.5 will be used to delete the record from GI. Figure 6.15 shows the GI after site 10 has been vanished.

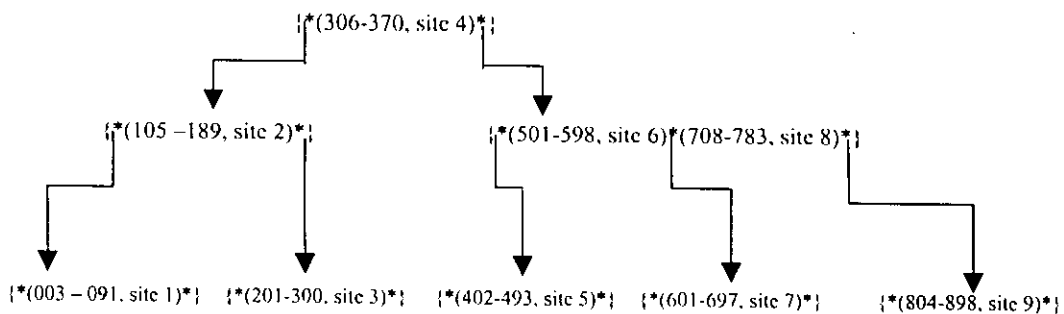


Figure 6.15: Global Index (Proposed B Tree of order 3) after Local Index at site 10 has been vanished

6.3.8 GI after site 8 has been vanished

When the records of site 8 is required to be deleted from the GI. We use the Algorithm 5.3 to find the record and the Algorithm 5.5 to delete the record. Figure 6.16 shows the GI after site 8 has been vanished.

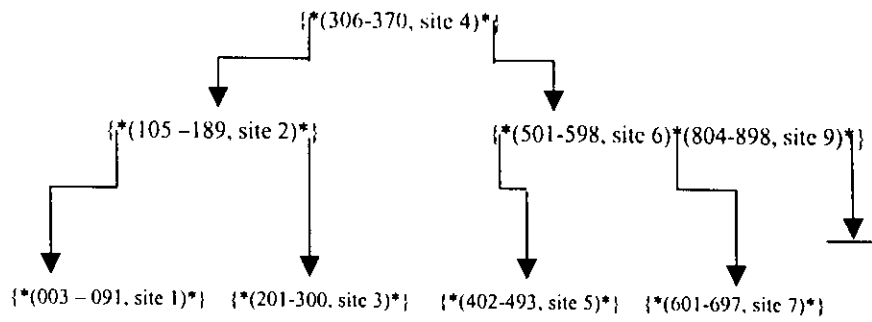


Figure 6.16: Global Index (Proposed B Tree of order 3) after Local Index at site 8 has been vanished

CONCLUSION

Although most of the databases are relational database at present other types of database like object oriented multimedia databases are now gaining popularity very much. Index structures like B, B*, B+ trees have proven performance on relational databases. Further study and experiment are required to see the performance of these trees on other types of databases. Actually, indexing in object oriented and multimedia databases introduces lot of new problems. These problems need to be solved with introducing new concepts and algorithms. We did not work on this issue in this thesis. But it is an interesting and demanding area to work under database research.

We proposed and examined the index algorithms for simple horizontally fragmented distributed databases. In fact, simple vertically fragmented distributed databases do not need to keep both LI and GI. In this case, the index-key would be either in one fragment or in more fragments and all the records (not with all the attributes due to vertical fragmentation) of the global relation will present in the fragment(s) and the LI in the fragment(s) alone will serve the purpose. Complex distributed databases having derived or mixed or both types of fragments will certainly require extra works to be done for indexing. More complex index structure and algorithms can be proposed and tested to get acceptable results.

Since there is no existing model of distributed index, we could not compare the performance of our proposed model against any model. New models can be proposed and compared with this model. This model proposed most of its algorithms based on previous works with minor variations. These variations do not have major performance effects. Further study can be conducted to get better algorithms.

In this model (distributed index), we did not consider the effects of network and system failure. But these are the important factors, which have serious effects on distributed databases as well as distributed index. Further study is required to improve the model to sustain and perform well even in the presence of these failures.

REFERENCES

- [1] J. Allen, M. Mcalling, "The Architecture of the Common Index Protocol (CIP)" (RFC draft version 1), 1997, [ftp://ftp.ietf.org/internet-drafts/draft-ietf-find-cip-arch-01.txt](http://ftp.ietf.org/internet-drafts/draft-ietf-find-cip-arch-01.txt)
- [2] Michael Abbey, Michael J. Corey, "ORACLE 8: A Beginner's Guide", Tata McGraw-Hill Publishing Company Limited, 1997.
- [3] G. M. Adel'son-Vel'skii and E. M. Landis, "An algorithm for the organization of information", Doklady Academia Nauk SSSR, vol. 146, no. 2, pp. 263-266, 1962. English translation in Soviet Mathematics, vol.3, no. 5, pp. 1259-1263, 1962.
- [4] The AltaVista Search Service, <http://www.altavista.digital.com/>
- [5] R. Bayer, E. McCreight "Organization and maintenance of a large ordered indexes", Acta Information, vol 1. No.3 pp.173-189, 1972.
- [6] M. Bowman, D. Hardy, M. Schwartz, D. Wessels, CIP Index Object Format for SOIF Objects (RFC draft version 2) April 1997, [ftp://ftp.ietf.org/internet-drafts/draft-ietf-find-cip-soif-02.txt](http://ftp.ietf.org/internet-drafts/draft-ietf-find-cip-soif-02.txt)
- [7] Stefano Ceri, Ginseppe Pelagatti, "Distributed Databases Principles & Systems", McGrawHill Book Company, 1984.
- [8] P. Deutsch, R. Schoultz, P. Faltstrom, C. Weider, Architecture of the Whois++ Index Service, RFC1835, August 1995, [ftp://ftp.ripe.net/rfc/rfc1835.txt](http://ftp.ripe.net/rfc/rfc1835.txt)
- [9] L. Gravano, K. Chang, H. Garcia-Molina, C. Lagoze, A. Paepcke, Stanford Protocol Proposal for Internet Search and Retrieval, January 1997, <http://www-db.stanford.edu/~gravano/starts.html/>
- [10] Free Harvest Web Indexing Software Development, <http://www.tardis.ed.ac.uk/harvest/>
- [11] The InfoSeek Search Service, <http://www.infoseek.com>
- [12] InfoSeek Distributed Search Pattern, 1997, http://software.infoseek.com/patents/dist_search/
- [13] D.E. Knuth, "The Art of Computer Programming vol.3: Sorting and Searching", Addison-Wesley, Reading Mass. 1973.
- [14] George Koch, Kevin Loney, "ORACLE The Complete Reference Third Edition" Osborne McGraw-Hill, 1995.
- [15] P. Panotzki, Complexity of the Common Indexing Protocol, September 1996, http://www.bunyip.com/research/papers/1996_cip_cip.html
- [16] M. Rio, J. Maccdo, V. Freitas, A Distributed Weighted Centroid-based Indexing System, in: Proceedings of the 8th European Networking Conference (JENC8), 1997, <http://www.terena.nl/conf/jenc8/papers/322.ps>
- [17] Abraham Silberschatz, Henry F. Korth, S. Sudarshan, "Database System Concepts", Third Edition, The McGraw Hill Companies, Inc, 1997.
- [18] Peter D. Smith, G. Michael Barnes, "Files and Databases: An Introduction", Addison-Wesley Publishing Company.
- [19] TERENA Task Force on Cooperative Hierarchical Indexing Coordination (TF-CHIC), <http://www.terena.nl/task-forces/tf-chic>

- [20] P. Valkenburg, D. Beckett, M. Hamilton, S. Wilkinson, Standards in the CHIC-Pilot Distributed Indexing Architecture, in: Computer Networks and ISDN Systems special issue " Proceedings of the TERENA Networking Conference 1998", <http://www.terena.nl/libr/tech/chic-fr.html/>
- [21] C. Weider, J. Fullton, S. Spero, Architecture of the Whois++ Index Service, RFC1913, February 1996, <ftp://ftp.ripe.net/rfc/rfc1913.txt>
- [22] M. Wahl, T. Howes, S. Kille, Lightweight Directory Access Protocol (v3), RFC2251, December 1997, <ftp://ftp.ripe.net/rfc/rfc2251.txt>
- [23] Stephen Wynkoop, Special Edition Using Microsoft SQL Server 6.5" Prentice-Hall of India Private Limited, 1998.
- [24] W3C's Distributed Indexing/Searching workshop, May 1996, <http://www.w3.org/Search/9605-Indexing-Workshop/>

