UNIT – VIII

8.1 Data on External Storage

The data stored on the database is so enormous that the main memory cannot accommodate. The data thus is stored on some external storage devices. Data is stored on external storage devices such as disks and tapes, and fetched into main memory as needed for processing. The unit of information read from or written to disk is a Page. The size of a page is DBMS parameter, and typical values are 4KB or 8KB.

The cost of page I/O dominates the cost of typical database operations, and database systems are carefully optimized to minimize this cost.

- Disks are the most important external storage devices. They allow us to retrieve any page at a fixed cost per page.
- Tapes are sequential access devices and force us to read data one page after the other. They are mostly used to archive data that is not needed on a regular basis.
- Each record in a file has a unique identifier called a Record ID. A rid has the property that we can identify the disk address of the page containing the record by using the rid.

Buffer Manager: Data is read into memory for processing, and written to disk for persistent storage, by a layer of software called the Buffer Manager. When the files and access methods layer needs to process a page, it asks the buffer manager to fetch the page, specifying the page's rid. The buffer manager fetches the page from disk if it is not already in memory.

Disk Space Manager: The Disk Space Manager manages Space on disk. When the records are coming to store in the files, the disk space manager allocate disk page for the file.

8.2 File Organization

File Organization is method of arranging records in files on the disk such that it computes maximum number of operation efficiently. However it is not possible to force efficient calculation for all operations. In DBMS, the file of records is an important abstraction. We can create a file, destroy it, and also can insert records into it and delete from it. It supports **scans** also. The scan operation allows us to step through all the records in the file one at a time. Typically a relation is stored as a file of records.

Heap File Organization: Any record can be stored at any place anywhere in the file, where there is space for the record. There is no ordering of records typically there is a single file for each relation. This is the simplest file structure. In a Heap File the records are stored in a random order across the pages of the file. Thus a file organization can be defined as the process of arranging the records in a file, when the file is stored on disk.

Sequential File Organization: Records are stored in a sequential order, according to the value of a search key for each record.

Hash File Organization: A hash function is computed on such attribute of each record. The result of the Hash function specifies in which block of the record should be placed.

Clustered File Organization: Records of several different relations are stored in the same file. Further related records of the different relations are stored on the same block. So that one input output relation fetches related records from all the relations.

8.2.1 Indexing

An index is a data structure, which organized data records on disk to optimize certain kinds of retrieval operations. Using an index we can easily retrieve the records, which satisfy search conditions on the search key fields of the index.

Data Entry: Which we use to refer to the records stored in an index file. We can search an index efficiently for finding desired data entries, and use them for obtaining data records.

The 3 Alternatives for Data Entries in an Index:

A data entry k* allows us to retrieve one or more data records with key value k. We need to consider three main alternatives:

1. A data entry k* is an actual data record (with search key value k).
2. A data entry is a <k, rid> pair, where rid is the record id of a data record with search key value k.
3. A data entry is a <k, rid-list> pair, where rid-list is a list of record ids of data records with search key value k.

If an index uses Alternative (1), there is no need to store the data records separately, in addition to the contents of the index. We can think of such an indexed file organization that can be used instead of a sorted file or a heap file organization.

The Alternatives (2) and (3), which contain data entries that point to data records, are independent of the file organization that is used for the indexed file. Alternative (3) offers better space utilization than Alternative (2), but data entries are variable in length, depending on the number of data records with a given search key value.

Search key: An attribute or set of attributes used to look up records in a fie is called a search key

8.2.2 Types of Indexes

Indexes enhance the performance of DBMS. They enable us to go to the desired record directly, without scanning each record in the file. The index page enables us to find the desired key word in the book, the need of sequential scan through the complete book. The index can be defined as a data structure that allows faster retrieval of data. Each index is based on the certain attribute of the field. This is given in search key. We can have several indexes based search keys.

Ordered Index:

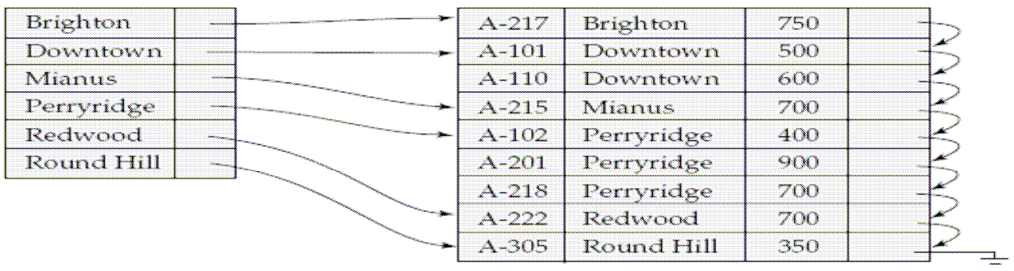
Based on a stored ordering of the values. To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted order, and associates with each search key the records that contain it. The records in the indexed file may themselves be stored in some sorted order, just as books in a library are stored according to some attribute such as the Dewey decimal number. A file may have several indices, on different search keys. If the file containing the records is sequentially ordered, a primary index is an index whose search key also defines the sequential order of the file. Primary indices are also called clustering indices. Indices whose search key specifies an order different from the sequential order of the file are called secondary indices, or non-clustering indices.

Primary Index:

Here we assume that all files are ordered sequentially on some search key. Such files, with a primary index on the search key, are called index-sequential files. They represent one of the oldest index schemes used in database systems. They are designed for applications that require both sequential processing of the entire file and random access to individual records. An index on a set of fields that includes the primary key is called a primary index. An index that is not a primary index is called a secondary index.

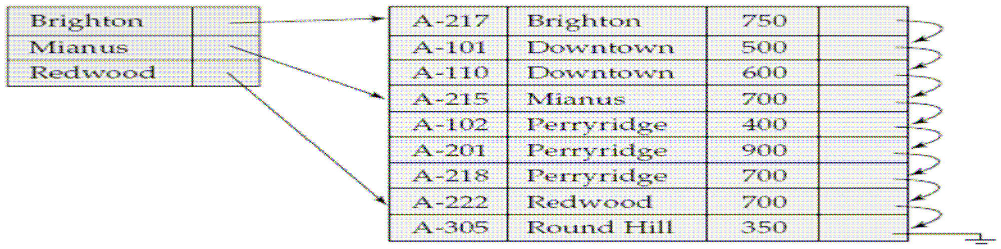
Dense index:

An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first record, since, because the index is a primary one, records are sorted on the same search key. Dense index implementations may store a list of pointers to all records with the same search-key value doing so is not essential for primary indices.



Sparse index:

An index record appears for only some of the search-key values. As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.

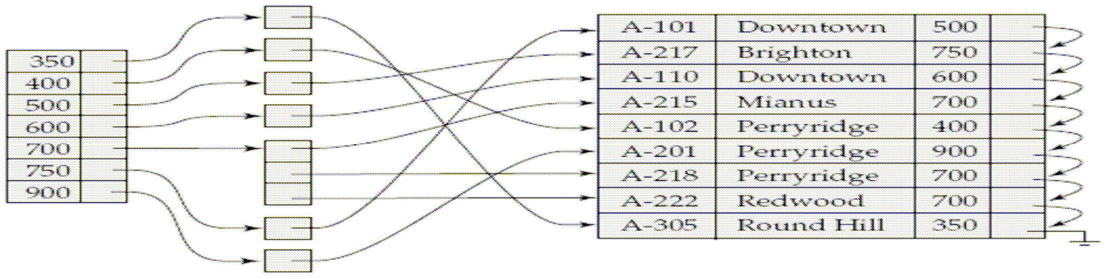


Brighton	
Mianus	
Redwood	

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Secondary Index:

A secondary index on a candidate key looks just like a dense primary index, except that the records pointed to by successive values in the index are not stored sequentially. In general, however, secondary indices may have a different structure from primary indices. If the search key of a primary index is not a candidate key, it suffices if the index points to the first record with a particular value for the search key, since the other records can be fetched by a sequential scan of the file. In contrast, if the search key of a secondary index is not a candidate key, it is not enough to point to just the first record with each search-key value. The remaining records with the same search-key value could be anywhere in the file, since the records are ordered by the search key of the primary index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.



350	
400	
500	
600	
700	
750	
900	

A-101	Downtown	500	
A-217	Brighton	750	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

Hash Index:

Based on uniform distribution of values across a range of buckets. The bucket to which a value is assigned is determined by a function called hash function. In a hash file organization we obtain the address of the disk block containing a desired record directly by computing a function on the search key value of the record. The bucket is to denote a unit of storage that can be store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or large than a disk block.

h(ki) = h(k4)

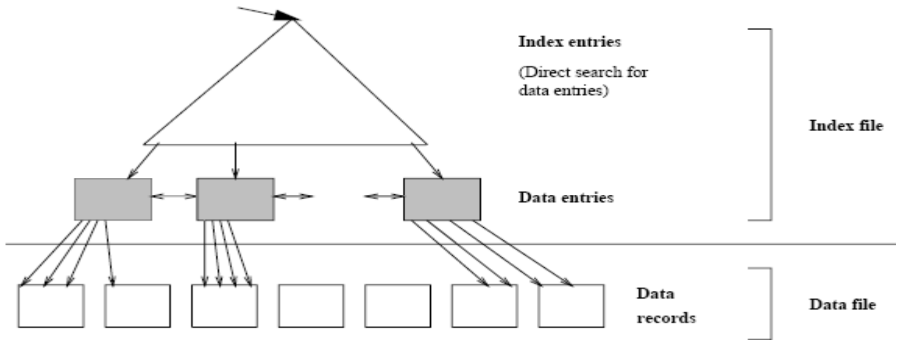
Where h = Hash function.

B = Set of bucket addresses

K = Set of search key.

<u>Clustered Index & Unclustered Index:</u>

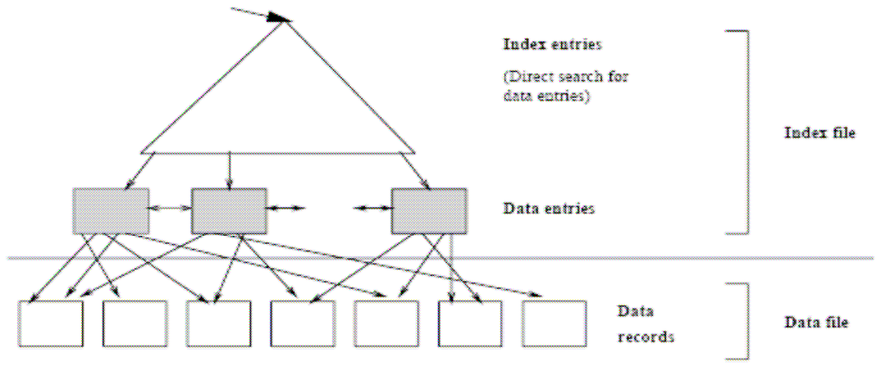
When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is clustered. Other wise it is Unclustered. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or Alternative (3) can be a clustered index only if the data records are sorted on the search key field. Indexes that maintain data entries in sorted order by search key use a collection of index entries, organized into a tree structure, to guide searches for data entries, which are stored at the leaf level of the tree in sorted order.



Clustered Index

A data file can be clustered on at most one search key, which means that we can have at most one clustered index on a data file. An index that is not clustered is called an Unclustered index; we can have several Unclustered indexes on a data file.

In practice, data records are rarely maintained in fully sorted order, unless data records are stored in an index-using Alternative (1), because of the high overhead of moving data records around to preserve the sort order as records are inserted and deleted. The cost of using an index to answer a range search query can vary tremendously based on whether the index is clustered. If the index is clustered, the rids in qualifying data entries point to a contiguous collection of records, as below Figure illustrates, and we need to retrieve only a few data pages. If the index is Unclustered, each qualifying data entry could contain a rid that points to a distinct data page, leading to as many data page I/Os as the number of data entries that match the range selection.



Unclustered index

Index Data Structures

The two methods are there, in which file data entries can be organized or arranged. They are

1. Hash – Based Indexing
2. Tree – Based Indexing.

1. Hash Based Indexing:

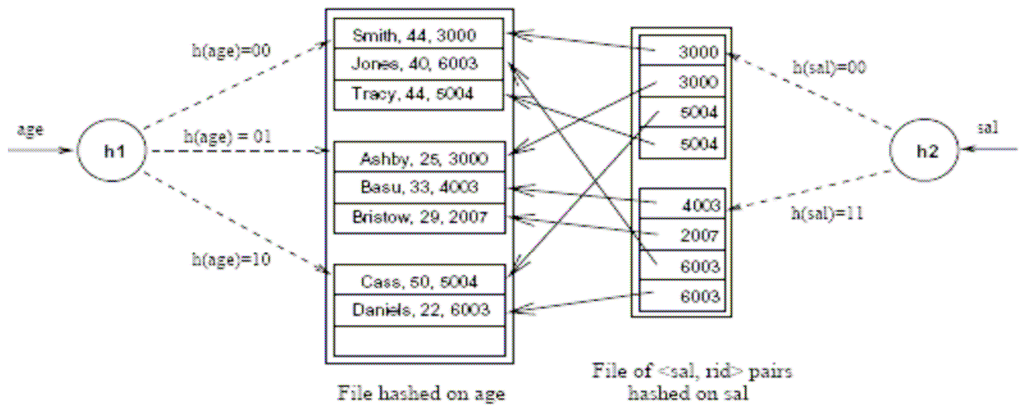
This type of indexing is used to find records quickly, by providing a search key value. We can organize records using a technique called hashing to quickly find records that have a given search key value. In this approach the records are grouped in Buckets. Here the Bucket contains the Primary page and additional pages linked in a chain. In order to determine to which Bucket a record belongs to a special function called Hash Function along with a search key is used.

Records insertion into Bucket: The records are inserted into the Bucket by allocating the needed OVER FLOW pages.

Record Searching: A hash function is used to find first, the bucket containing the records and then by scanning all the pages in a bucket, the record with a given search key can be found. Here if the record doesn't have search key value then all the pages in the file needs to be scanned.

Record Retrieval: By applying a hash function to the record's search key, the page containing the needed record can be identified and retrieved in one disk I/O.

Example: Consider a file Employee that represents data records with a hash key age. Applying the hash function to the age represents page that contains the needed record. The hash function h converts the search key value to its binary representation and used the two least significant bits as the bucket identifier. Below figure shows that the index with search key sal that contains <sal, rid> pairs as data entries. The record id component of a data entry in this second index is a pointer to a record with search key value sal. Note that the search key for an index can be any sequence of one or more fields, and it need not uniquely identify records. For example, in the salary index, two data entries have the same search key value 6003.



File hashed on Age, with Index on Sal

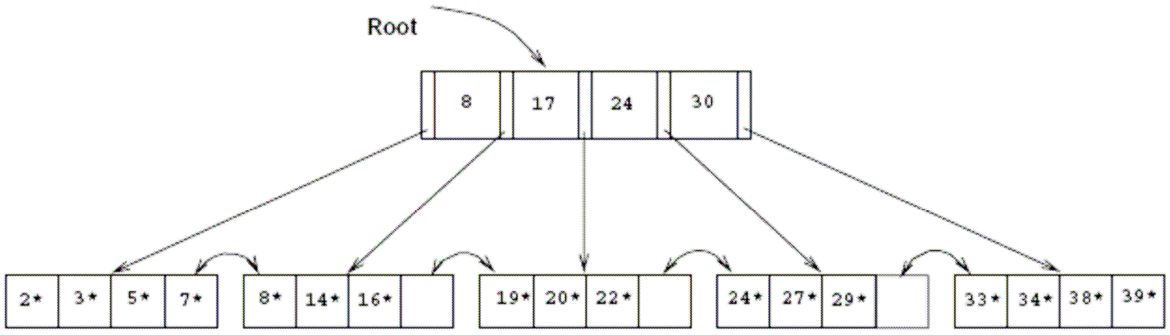
2. Tree – Based Indexing:

An alternative method to hash – based indexing is to organize the records using a tree – liked data structure. The records are arranged in a Tree – like structure here. The data entries are sorted according to the search key values and they are arranged in a hierarchical manner to find the correct page of the data entries.

Example:

Below the figure shows that the employee records from the above figure this time organized in a tree – structured index with search key age. Each node in this figure (Ex: A, B, L1, L2) is a physical page, and retrieving a node involves a disk I/o.

The lowest level of the tree, called the **leaf level**, contains the data entries here these are the employee records. Here the additional records are to be inserted with age less than 22 and age greater than 50, left side of the L1 and to the right of the leaf node L3.



Tree – Structured Index

All searches begin at the topmost node, called the **Root Node**. The Non – Leaf pages contain node pointers separated by search key values. The node pointer to the left of a key value k points to a subtree that contains only data entries less than k. The node pointer to the right of a key value k points to a subtree that contains only data entries greater than or equal to k.

Comparison of File Organizations

We now compare the costs of some simple operations for several basic file organizations on a collection of records. Here we assume that the files and indexes are organized according to the composite search key and that all selection operations are specified on these fields.

For sorted and hashed files, the sequence of fields on which the file is sorted or hashed is called the **search key**.

Scan: Fetch all records in the file. The pages in the file must be fetched from disk into the buffer pool. There is also a CPU overhead per record for locating the record on the page.

Search with equality selection: Fetch all records that satisfy an equality selection, for example, “Find the Employees record for the Employee with age 23 and Sal 50" Pages that contain qualifying records must be fetched from disk, and qualifying records must be located within retrieved pages.

Search with range selection: Fetch all records that satisfy a range selection, for example, “Find all Employee records with name alphabetically after `Smith.' "

Insert: Insert a given record into the file. We must identify the page in the file into which the new record must be inserted, fetch that page from disk, modify it to include the new record, and then write back the modified page. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Delete: Delete a record that is specified using its rid. We must identify the page that contains the record, fetch it from disk, modify it, and write it back. Depending on the file organization, we may have to fetch, modify, and write back other pages as well.

Cost Model:

It is a method used to calculate the costs of different operations that are performed on the database. That is in terms of time needed for execution.

Notations used in this model:

B = The total no. of pages with out any space wastage when records are grouped into it.

R = The total no. of records present in a page.

D = The average time needed to Read or Write a disk page.

C = The average time needed for a Record processes.

H = Time required the application of hash function on a record in hashed file organization.

F = Fan – out (in Tree Indexes).

For calculating input and output costs, which is the base for cost of the database operations. We take, D = 15ms, C and H = 100ns.

Observation to be Note:

1. CPU Costs are considered in most of the real systems.
2. In order to reduce the complexity of the cost model, the no. of pages that are read from or written to the disk are counted for finding input and output cost but the problem with this approach is that it ignores **blocked access**.
3. The cost is equal to the time required to **seek** the first page in the block and to **transfer** all pages in the block. Such blocked access can be much cheaper than issuing one I/O request per page in the block, especially if these requests do not follow consecutively: We would have an additional seek cost for each page in the block.

1. Heap Files:

<u>Cost of Scanning:</u> The cost of scanning heap files is B (D + RC). That is scanning R records of B pages with time C per record would take BRC and scanning B pages with time D per page would take BD. Hence the total cost incurred in scanning is

$$BD + BRD \Rightarrow B(D+RC)$$

<u>Search with equality selection:</u> Searching exactly one record that meets the quality criteria involves scanning half of the files based on the assumption that a record exists in that part would take,

$$½ * \text{Scanning Cost} \Rightarrow ½ * B(D+RC)$$

<u>Search with range selection:</u> This is the cost incurred in of scanning because it is not known in advance how many records can satisfy the particular range. Hence we need to scan the entire file that would take Cost of searching the records with range selection = Cost of scanning.

$$B(D+RC)$$

<u>Cost of Insertion</u>: The cost incurred in inserting a record in a heap file is given as 2D + C. Because for inserting a record we need to fetch the last page of the file that would take time D. Then we need to add the record that takes time C. And finally the page is written back to the disk in time D. Hence the total cost is

$$D + D + C \Rightarrow 2D + C$$

<u>Cost of Deletion:</u> In order to delete record we need to search the record by reading the page that would take time D. Then the Record is removed in time C and finally the modified page is written back to the disk in time D. Hence the total time is 2D + C. That is the cost incurred in deleting a record from a heap file is given as, Cost of Searching + C + D ⇒ D + D + C ⇒ 2D + C

<u>2. Sorted Files:</u>

<u>Cost of Scan:</u> The cost of scanning sorted files is B (D + RC). Because all the pages need to be scanned in order to retrieve a record. That is

Cost of Scanning Sorted files = Cost of scanning Heap Files

<u>Search with equality selection:</u> This cost in sorted files equal to $D \log_2 B + C \log_2 R$. It improves the performance and is the sum of, $D \log_2 B$ = It is the time required for performing a binary search for a page that contains the qualifying record and

$C \log_2 B$ = It is the time required for performing a binary search to find the first satisfying record in a page. If many records satisfies, that record is equal to $D \log_2 B + C \log_2 R$ + Cost of sequential reading of all the qualifying records.

<u>Search with range selection:</u> This cost is given as Cost of fetching the first matching records page + cost of obtaining the set of qualifying records. If the range is small then a single page contain all the matching records else, additional pages needs to be fetched.

<u>Cost of Insertion:</u> The cost of insertion in sorted files is given by Search cost + B(D + RC)

Which includes, finding correct position of the record + adding of record + fetching of pages + Rewriting the pages that is, search cost plus B (D + RC).

Cost of Deletion: The cost of deletion in sorted files is given by Search cost + B(D+RC)

Which is same as the cost of insertion, and includes

Searching a Record + Removing Record + Rewriting the modified page.

3. Clustered Files:

Cost of Scan: The cost of scanning clustered files is the same as the cost of sorted files expect that it have more no. of pages and this cost is given as

Scanning B pages with D per page takes BD and scanning R records of B pages with time C per records take BRC. The total cost is 1.5 B(D+RC)

Search with equality selection:

1. **For a single qualifying record:** The cost incurred in finding a single qualifying record in clustered files is the sum of the binary searches involved in finding the first page in D $\log_F$ 1.5 B and finding the first matching record in C $\log_2$ R. Hence it is given as D $\log_F$ 1.5 B + C $\log_2$ R
2. **For several Qualifying Records:** If more than one record satisfies the selection criteria then they are assumed to be located consequently hence, the cost required in finding such records is equal to D logF 1.5 B + C log2 R + Cost involved in sequential reading in all records.

Search with range selection: This cost is same as the cost incurred in an equality search under several matched records. In this, first satisfying record is identified followed by the sequential retrieval of the next pages until a non-qualifying record is found.

Cost of Insertion: The cost of insertion in clustered files is given by Search + write $\Rightarrow$ (D $\log_F$ 1.5 B + C $\log_2$ R) + D. The correct leaf page having an empty space for a new record is first founded by traversing from a root node to the leaf; the new record is then inserted followed by the writing of the modified page.

Cost of Deletion: It is same as the cost of insertion and includes,

The cost of searching for a record + removing of a record + rewriting the modified page.

$\Rightarrow$ (D $\log_F$ 1.5 B + C $\log_2$ R) + D

I/O Costs Comparison

File Type	Scan	Equality Search	Range Search	Insert	Delete
Heap	B(D+RC)	½ B(D+RC)	B(D+RC)	2D+C	2D+C
Sorted	B(D+RC)	D $\log_2$ B + C	D $\log_2$ B + #	Search cost +	Search cost +

		$\log_2 R$	Matches	B(D+RC)	B(D+RC)
Clustered	1.5 B(D+RC)	D $\log_F$ 1.5 B + C $\log_2 R$	D $\log_F$ 1.5 B + # Matches	(D $\log_F$ 1.5 B + C $\log_2 R$) + D	(D $\log_F$ 1.5 B + C $\log_2 R$) + D

Choosing a File Organization:

The above table compares I/O costs for the file organizations. A heap file has good storage efficiency and supports fast scan, insertion, and deletion of records. However, it is slow for searches.

A sorted file also offers good storage efficiency, but insertion and deletion of records is slow. It is quite fast for searches, and it is the best structure for range selections. It is worth noting that in a real DBMS, a file is almost never kept fully sorted. Files are sometimes kept `almost sorted' in that they are originally sorted, with some free space left on each page to accommodate future insertions, but once this space is used, overflow pages are used to handle insertions. The cost of insertion and deletion is similar to a heap file, but the degree of sorting deteriorates as the file grows.

A hashed file does not utilize space quite as well as a sorted file, but insertions and deletions are fast, and equality selections are very fast. However, the structure offers no support for range selections, and full file scans are a little slower; the lower space utilization means that files contain more pages.

The above table demonstrates that no one-file organization is uniformly superior in all situations. An unordered file is best if only full file scans are desired. A hashed file is best if the most common operation is an equality selection. A sorted file is best if range selections are desired. The organizations that we have studied here can be improved on the problems of overflow pages in static hashing can be overcome by using dynamic hashing structures, and the high cost of inserts and deletes in a sorted file can be overcome by using tree-structured indexes but the main observation, that the choice of an appropriate file organization depends on how the file is commonly used, remains valid.

Indexes and Performance Tuning

The performance of a system depends greatly on the indexes we choose and can be explained in terms of the expected workload.

Workload impact: Data entries that qualify particular selection criteria can be retrieved effectively by means of indexes. Two selection types are

1. Equality Selection
2. Range Selection

Tree based indexing supports both the selection criteria as well as inserts, deletes and updates where as only quality selection is supported by hash based indexing apart from insertion, deletion, and updation.

Advantages of using Tree structured indexes:

1. By using tree-structured indexes, insertion and deletion of data entries can be handled effectively.
2. It finds the correct leaf page faster than binary search in sorted files.

Disadvantages:

The sorted file pages are in accordance with the disks order hence sequential retrieval of such pages is quicker which is not possible in tree – structured indexes. This drawback can be overcome by the use of ISAM that provides fast searching along with the sequential allocation of the leaf pages.

Clustered index organization:

We must have at least one clustered index in addition to several Unclustered indexes in order to prevent the duplication of large data records.

Example: The search key sno in student record is clustered index where as marks is an Unclustered index.

Using clustered indexes for indexing cheap and an addition of a new record in a leaf page that is full causes the creation of a new page with the assignment of some old records to that new page. All the database pointers must new point to the new page, which involves many disk I/Os. Hence, it is used rarely.

Index only-evaluation:

As the name implies the query evaluation here is done solely through the file indexes rather than accessing all the data records.

Advantages: It uses only Unclustered indexes.

Example: If we want to find out the average marks obtained by the students in an exam a part from having an index and marks, then DBMS finds this by using index data entries.

Clustered Indexes-Example:

Consider the following the database query:

SELECT s.rno from student s

WHERE s.marks > 70.

If B+ tree index on rno exists then all the students whose marks > 70 can be retrieved but this depends on the condition and the no. of students whose scored marks > 70. Two cases arises.

Case 1: If all the students obtained marks > 70 then sequential scanning is advantageous.

Case 2: If only few students secured > 70 marks then it depends on the type of index.

If it is an Unclustered then it involves one I/O for qualifying student. Which would be expensive else if it is a clustered index then it requires only 10% of the I/Os incurred in scan.

Use of Aggregate Operators:

Consider the given example

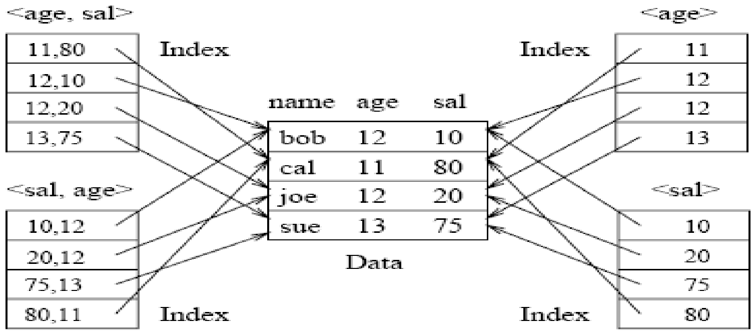
SELECT s.sno, COUNT(*) from student s GROUP BY s.sno.

The purpose of this query is to count the no. of students in each section, if hash or B+ tree index exists on sno then scanning is appropriate. For each value of sno we count the no. of indexes data entries. The type of index does not matter because of the absence of the retrieval operation.

Composite Search Keys:

The search key for an index can contain several fields; such keys are called composite search keys or concatenated keys. As an example, consider a collection of employee records, with fields name, age, and sal, stored in sorted order by name. Below Diagram illustrates the difference between a composite index with key <age, sal>, a composite index with key hsal, agei, an index with key age, and an index with key sal.

If the search key is composite, an equality query is one in which each field in the search key is bound to a constant. For example, we can ask to retrieve all data entries with age = 20 and sal = 10. The hashed file organization supports only equality queries, since a hash function identifies the bucket containing desired records only if a value is specified for each field in the search key. A range query is one in which not all fields in the search key are bound to constants. For example, we can ask to retrieve all data entries with age = 20; this query implies that any value is acceptable for the sal field. Example of a range query, we can ask to retrieve all data entries with age < 30 and sal > 40.



Composite Search key Index

Index Specification In SQL:

The SQL standard does not include any statement for creating or dropping index structures. In fact, the standard does not even require SQL implementations to support indexes! In practice, of course, every commercial relational DBMS supports one or more kinds of indexes. The following command to create a B+ tree index.

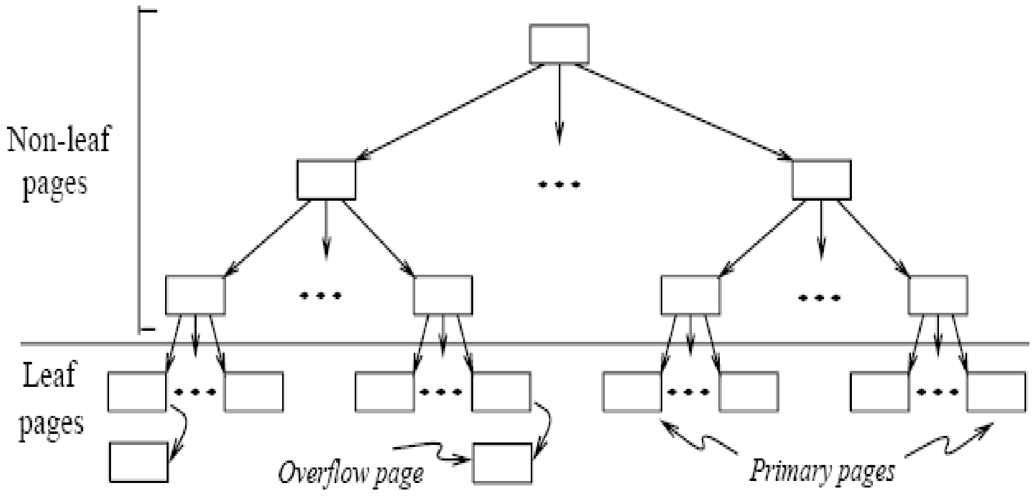
```
CREATE INDEX IndAgeRating ON Students
WITH STRUCTURE = BTREE,
KEY = (age, gpa)
```

This specifies that a B+ tree index is to be created on the Students table using the concatenation of the age and gpa columns as the key. Thus, key values are pairs of the form <age, gpa>, and there is a distinct entry for each such pair. Once the index is created, it is automatically maintained by the DBMS adding/removing data entries in response to inserts/deletes of records on the Students relation.

8.7 Indexed Sequential Access Methods (ISAM)

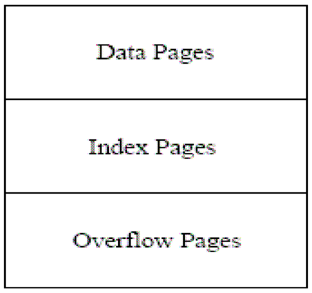
The ISAM is static method. To understand the motivation for the ISAM technique, it is useful to begin with a simple sorted file.

The data entries of the ISAM index are in the leaf pages of the tree and additional overflow pages that are chained to some leaf page. In addition, some systems carefully organize the layout of pages so that page boundaries correspond closely to the physical characteristics of the underlying storage device. The ISAM structure is completely static and facilitates such low-level optimizations.



Structure of ISAM

Each tree node is a disk page, and all the data resides in the leaf pages. When the file is created, all leaf pages are allocated sequentially and sorted on the search key value. The non-leaf level pages are then allocated. If there are several inserts to the file subsequently, so that more entries are inserted into a leaf than will fit onto a single page, additional pages are needed because the index structure is static. These additional pages are allocated from an overflow area. The allocation of pages is illustrated in below Figure.



Page Allocation in ISAM

The basic operations of insertion, deletion, and search are all quite straightforward. For an equality selection search, we start at the root node and determine which subtree to search by comparing the value in the search field of the given record with the key values in the node. For a range query, the starting point in the data level is determined similarly, and data pages are then retrieved sequentially.

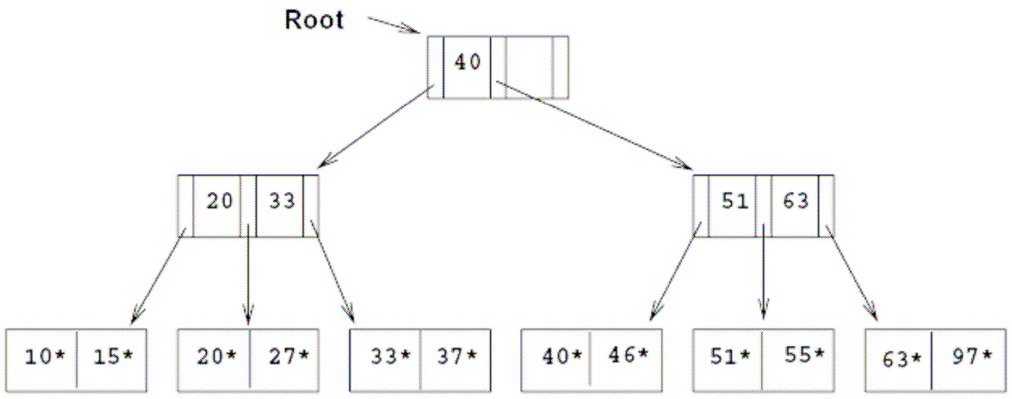
For inserts and deletes, the appropriate page is determined as for a search, and the record is inserted or deleted with overflow pages added if necessary.

The following example illustrates the ISAM index structure. Consider the tree shown in below Figure. All searches begin at the root. For example, to locate a record with the key value 27, we start at the root and follow the left pointer, since 27 < 40. We then follow the middle pointer, since 20 <= 27 < 33. For a range search, we find the first qualifying data entry as for an equality selection and then retrieve primary leaf pages sequentially. The primary leaf pages are assumed to be allocated sequentially this assumption is reasonable because the number of such pages is known when the tree is created and does not change subsequently under inserts and deletes and so no “next leaf page” pointers are needed.

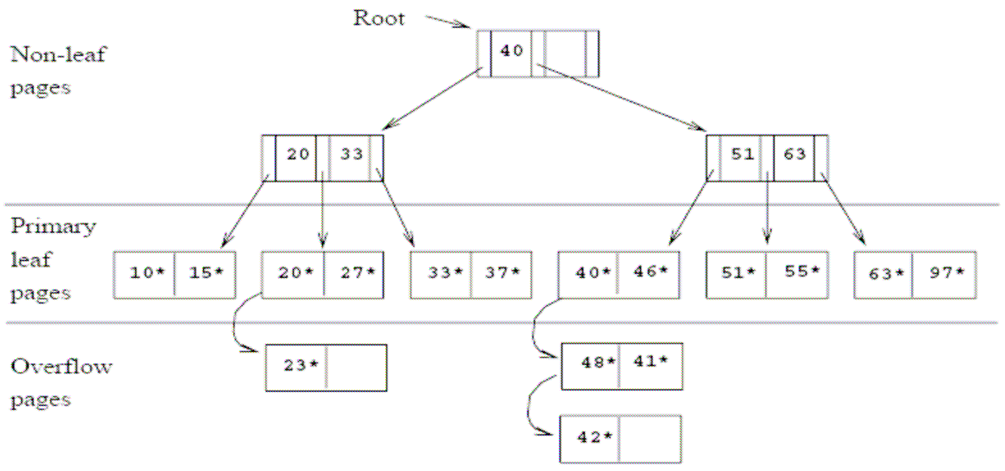
We assume that each leaf page can contain two entries. If we now insert a record with key value 23, the entry 23* belongs in the second data page, which already contains 20* and 27* and has no more space. We deal with this situation by adding an overflow page and putting 23* in the overflow page. Chains of overflow pages can easily develop. For instance, inserting 48*, 41*, and 42* leads to an overflow chain of two pages.

The deletion of an entry k* is handled by simply removing the entry. If this entry is on an overflow page and the overflow page becomes empty, the page can be removed. If the entry is on a primary page and deletion makes the primary page empty. Thus, the number of primary leaf pages is fixed at file creation time. Notice that deleting entries could lead to a situation in which key values that appear in the index levels do not appear in the leaves Since index levels are used only to direct a search to the correct leaf page, this situation is not a problem. Note that after deleting 51*, the key value 51 continues to appear in the index level. A subsequent search for 51* would go to the correct leaf page and determine that the entry is not in the tree.

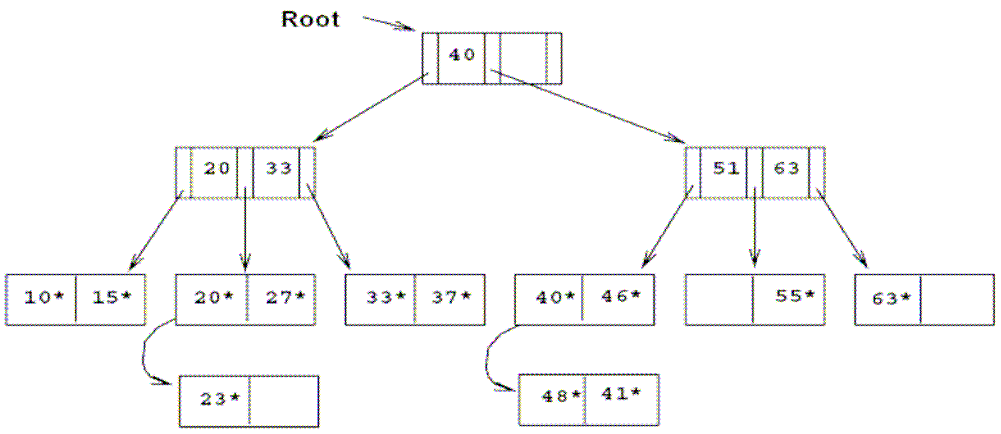
The non-leaf pages direct a search to the correct leaf page. The number of disk I/Os is equal to the number of levels of the tree and is equal to logf N, where N is the number of primary leaf pages and the **fan-out** F is the number of children per index page. This number is considerably less than the number of disk I/Os for binary search, which is log2N; in fact, pinning the root page in memory reduces it further. The cost of access via a one-level index is log2(N=F). If we consider a file with 1,000,000 records, 10 records per leaf page, and 100 entries per index page, the cost of a file scan is 100,000, a binary search of the sorted data file is 17, a binary search of a one-level index is 10, and the ISAM file is 3.



Sample ISAM Tree



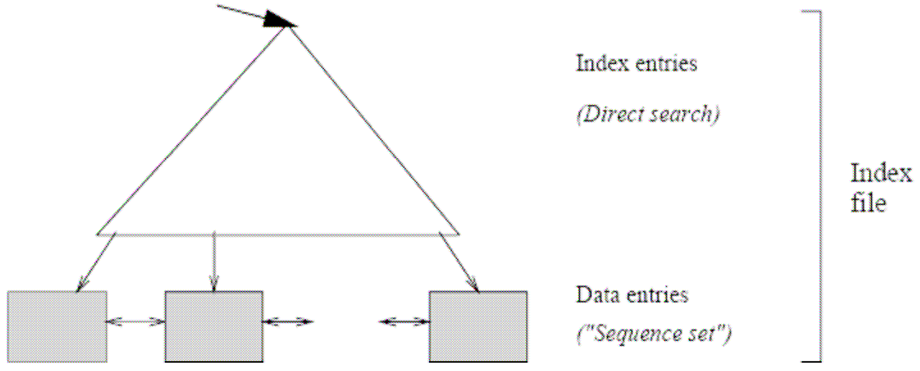
ISAM Tree after Inserts

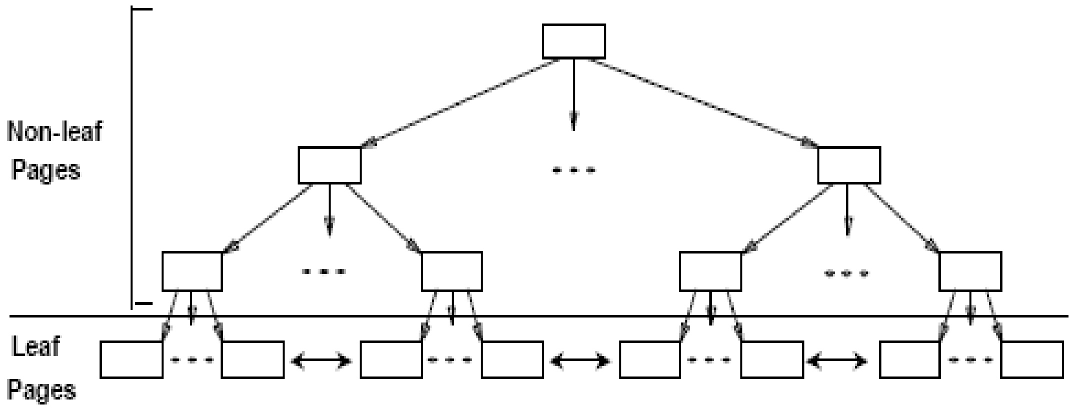


ISAM Tree after Deletes

8.8 B+ Trees (A Dynamic Data Structure)

A static structure such as the ISAM index suffers from the problem that long overflow chains can develop as the file grows, leading to poor performance. This problem motivated the development of more flexible, dynamic structures that adjust gracefully to inserts and deletes. The **B+ tree** search structure, which is widely used, is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries. Since the tree structure grows and shrinks dynamically. In order to retrieve all leaf pages efficiently, we have to link them using page pointers. By organizing them into a doubly linked list, we can easily traverse the sequence of leaf pages sometimes called the **sequence set** in either direction. The Structure of the B+ tree is below.





Structure of a B+ Tree

The following are some of the main characteristics of a B+ tree:

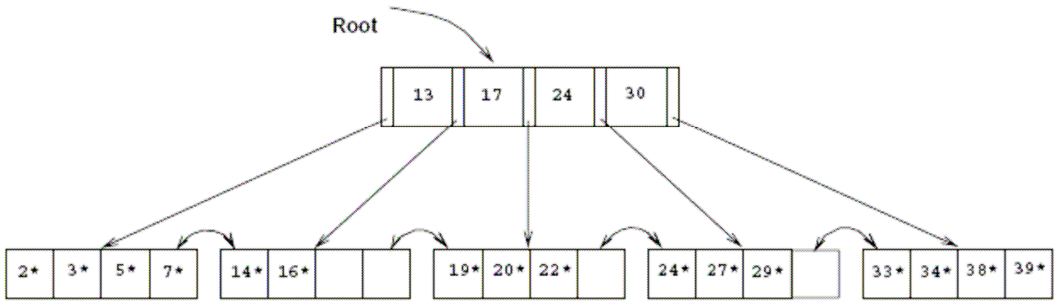
1. Operations (insert, delete) on the tree keep it balanced.
2. A minimum occupancy of 50 percent is guaranteed for each node except the root if the deletion algorithm is implemented. However, deletion is often implemented by simply locating the data entry and removing it, without adjusting the tree as needed to guarantee the 50 percent occupancy, because files typically grow rather than shrink.
3. Searching for a record requires just a traversal from the root to the appropriate leaf. We will refer to the length of a path from the root to a leaf any leaf, because the tree is balanced as the **height** of the tree.
4. The B+ trees in which every node contains m entries, where d <= m <= 2d. The value d is a parameter of the B+ tree, called the **order** of the tree, and is a measure of the capacity of a tree node. The root node is the only exception to this requirement on the number of entries; for the root it is simply required that 1 <= m <= 2d.

SEARCH:

The algorithm for search finds the leaf node in which a given data entry belongs. We use the notation *ptr to denote the value pointed to by a pointer variable ptr and & to denote the address of value. Note that finding i in tree search requires us to search within the node, which can be done with either a linear search or a binary search.

In discussing the search, insertion, and deletion algorithms for B+ trees, we will assume that there are no duplicates. That is, no two data entries are allowed to have the same key value. Of course, duplicates arise whenever the search key does not contain a candidate key and must be dealt with in practice.

Consider the sample B+ tree shown in below Figure. This B+ tree is of order d=2. That is, each node contains between 2 and 4 entries. Each non-leaf entry is a <key value, node pointer> pair; at the leaf level, the entries are data records that we denote by k*. To search for entry 5*, we follow the left-most child pointer, since 5 < 13. To search for the entries 14* or 15*, we follow the second pointer, since 13 < 14 < 17, and 13 < 15 < 17. To find 24*, we follow the fourth child pointer, since 24 < 24 < 30.



INSERT:

The algorithm for insertion takes an entry, finds the leaf node where it belongs, and inserts it there. The basic idea behind the algorithm is that we recursively insert the entry by calling the insert algorithm on the appropriate child node. Usually, this procedure results in going down to the leaf node where the entry belongs, placing the entry there, and returning all the way back to the root node. Occasionally a node is full and it must be split. When the node is split, an entry pointing to the node created by the split must be inserted into its parent; this entry is pointed to by the pointer variable newchildentry. If the root is split, a new root node is created and the height of the tree increases by one.

To illustrate insertion, if we insert entry 8*, it belongs in the left-most leaf, which is already full. This insertion causes a split of the leaf page; the split pages are shown in Figure 1. The tree must now be adjusted to take the new leaf page into account, so we insert an entry consisting of the pair <5, pointer to new page> into the parent node. Notice how the key 5, which discriminates between the split leaf page and its newly created sibling, is `copied up.' We cannot just `push up' 5, because every data entry must appear in a leaf page.

Since the parent node is also full, another split occurs. In general we have to split a non-leaf node when it is full, containing 2d keys and 2d + 1 pointers, and we have to add another index entry to account for a child split. We now have 2d + 1 keys and 2d+2 pointers, yielding two minimally full non-leaf nodes, each containing d keys and d+1 pointers, and an extra key, which we choose to be the `middle' key. This key and a pointer to the second non-leaf node constitute an index entry that must be inserted into the parent of the split non-leaf node. The middle key is thus `pushed up' the tree, in contrast to the case for a split of a leaf page. The split pages in our example are shown in Figure 9.2. The index entry pointing to the new non-leaf node is the pair <17, pointer to new index-level page>; notice that the key value 17 is `pushed up' the tree, in contrast to the splitting key value 5 in the leaf split, which was `copied up.'

The difference in handling leaf-level and index-level splits arises from the B+ tree requirement that all data entries kfi must reside in the leaves. This requirement prevents us from `pushing up' 5 and leads to the slight redundancy of having some key values appearing in the leaf level as well as in some index level. However, range queries can be efficiently answered by just retrieving the sequence of leaf pages; the redundancy is a small price to pay for efficiency. In dealing with the index levels, we have more flexibility, and we `push up' 17 to avoid having two copies of 17 in the index levels.

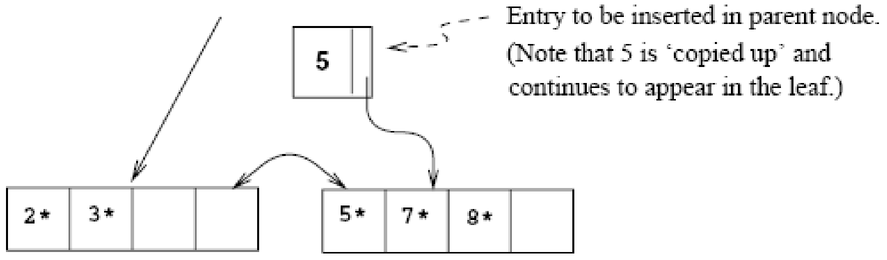

Entry to be inserted in parent node.
(Note that 5 is 'copied up' and
continues to appear in the leaf.)
5
2*
3*
5*
7*
8*

Figure 1 Split Leaf Pages during Insert of Entry 8*

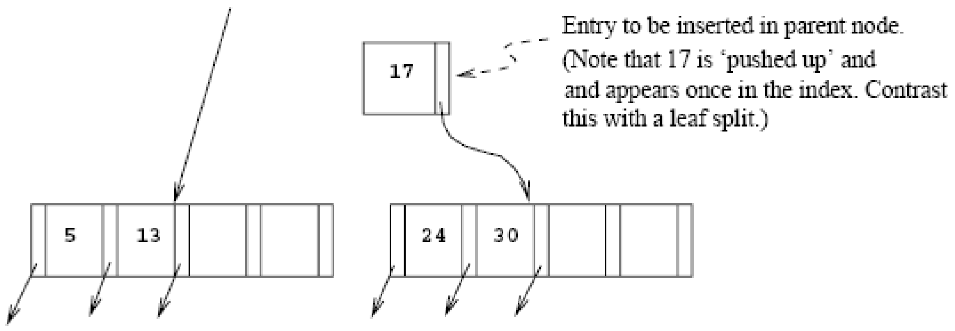

Entry to be inserted in parent node.
(Note that 17 is 'pushed up' and
and appears once in the index. Contrast
this with a leaf split.)
17
5
13
24
30

Figure 2 Split Index Pages during Insert of Entry 8*

Now, since the split node was the old root, we need to create a new root node to hold the entry that distinguishes the two split index pages. The tree after completing the insertion of the entry 8* is shown in Figure 9.3.

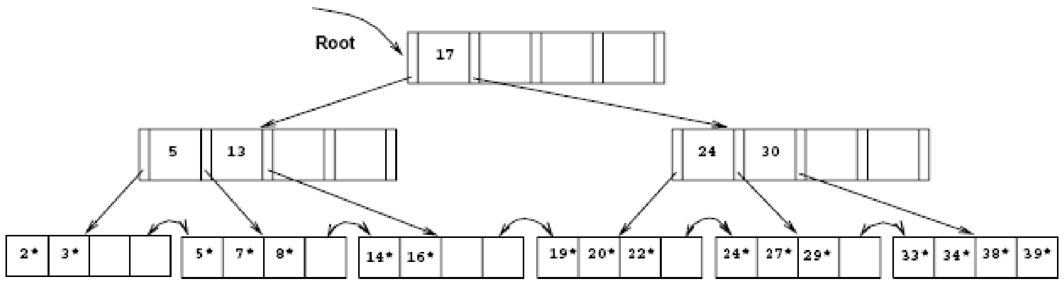


Figure 3 B+ Tree after Inserting Entry 8*

The entry belongs in the left-most leaf, which is full. However, the sibling of this leaf node contains only two entries and can thus accommodate more entries. We can therefore handle the insertion of 8* with a redistribution. Note how the entry in the parent node that points to the second leaf has a new key value; we `copy up' the new low key value on the second leaf. This process is illustrated in Figure 9.4.

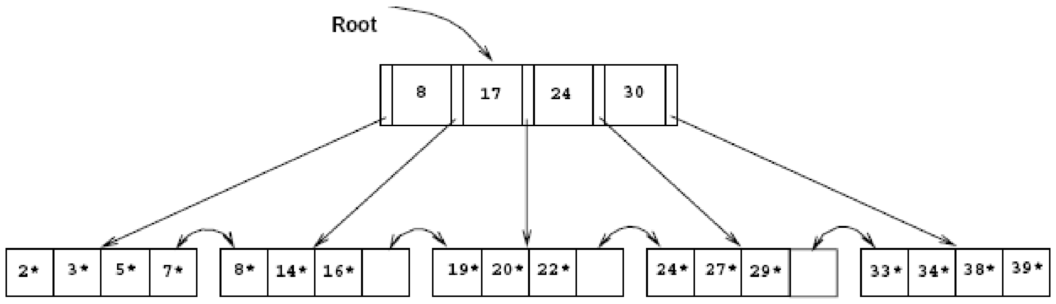


Figure 4 B+ Tree after Inserting Entry 8* Using Redistribution

To determine whether redistribution is possible, we have to retrieve the sibling. If the sibling happens to be full, we have to split the node anyway. On average, checking whether redistribution is possible increases I/O for index node splits, especially if we check both siblings. If the file is growing, average occupancy will probably not be affected much even if we do not redistribute. Taking these considerations into account, not redistributing entries at non-leaf levels usually pays off.

If a split occurs at the leaf level, however, we have to retrieve a neighbor in order to adjust the previous and next-neighbor pointers with respect to the newly created leaf node. Therefore, a limited form of redistribution makes sense: If a leaf node is full, fetch a neighbor node; if it has space, and has the same parent, redistribute entries. Otherwise split the leaf node and adjust the previous and next-neighbor pointers in the split node, the newly created neighbor, and the old neighbor.

DELETE:

The algorithm for deletion takes an entry, finds the leaf node where it belongs, and deletes it. The basic idea behind the algorithm is that we recursively delete the entry by calling the delete algorithm on the appropriate child node. We usually go down to the leaf node where the entry belongs, remove the entry from there, and return all the way back to the root node. Occasionally a node is at minimum occupancy before the deletion, and the deletion causes it to go below the occupancy threshold. When this happens, we must either redistribute entries from an adjacent sibling or merge the node with a sibling to maintain minimum occupancy. If entries are redistributed between two nodes, their parent node must be updated to reflect this; the key value in the index entry pointing to the second node must be changed to be the lowest search key in the second node. If two nodes are merged, their parent must be updated to reflect this by deleting the index entry for the second node; this index entry is pointed to by the pointer variable oldchildentry when the delete call returns to the parent node. If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree decreases by one.

To delete entry 19*, we simply remove it from the leaf page on which it appears, and we are done because the leaf still contains two entries. If we subsequently delete 20*, however, the leaf contains only one entry after the deletion. The sibling of the leaf node that contained 20* has three entries, and we can therefore deal with the situation by redistribution; we move entry 24* to the leaf page that contained 20* and `copy up' the new splitting key into the parent. This process is illustrated in Figure 7.

Suppose that we now delete entry 24*. The affected leaf contains only one entry (22*) after the deletion, and the sibling contains just two entries (27* and 29*). Therefore, we cannot redistribute entries. However, these two leaf nodes together contain only three entries and can be merged. While merging, we can `toss' the entry (h27, pointer to second leaf pagei) in the parent, which pointed to the second leaf page, because the second leaf page is empty after the merge and can be discarded. The right subtree of Figure 7 after this step in the deletion of entry 24* is shown in Figure 8.

Deleting the entry <27, pointer to second leaf page> has created a non-leaf-level page with just one entry, which is below the minimum of d=2. To fix this problem, we must either redistribute or merge. In either case we must fetch a sibling. The only sibling of this node contains just two entries (with key values 5 and 13), and so redistribution is not possible; we must therefore merge.
The situation when we have to merge two non-leaf nodes is exactly the opposite of the situation when we have to split a non-leaf node. We have to split a non-leaf node when it contains 2d keys and 2d + 1 pointers, and we have to add another key pointer pair. Since we resort to merging two non-leaf nodes only when we cannot redistribute entries between them, the two nodes must be minimally full; that is, each must contain d keys and d+1 pointers prior to the deletion. After merging the two nodes and removing the key pointer pair to be deleted, we have 2d−1 keys and 2d+1 pointers: Intuitively, the left-most pointer on the second merged node lacks a key value. To see what key value must be combined with this pointer to create a complete index entry, consider the parent of the two nodes being merged. The index entry pointing to one of the merged nodes must be deleted from the parent because the node is about to be discarded.

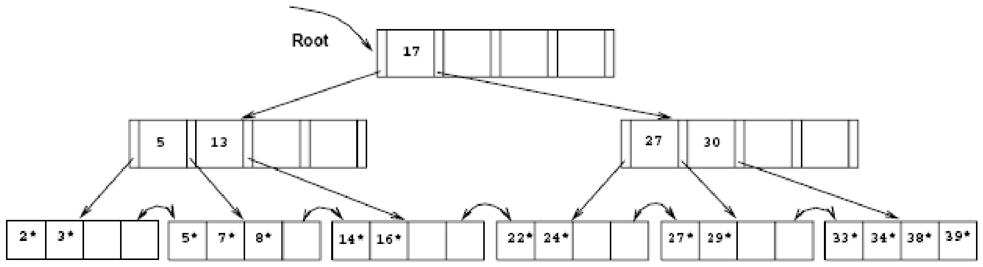


Figure 7 B+ Tree after Deleting Entries 19* and 20*

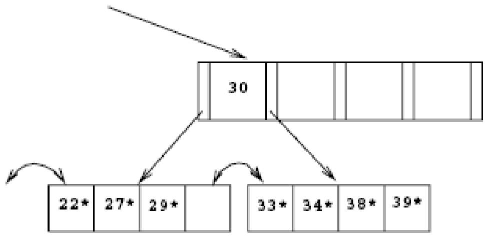


Figure 8 Partial B+ Tree during Deletion of Entry 24*

The key value in this index entry is precisely the key value we need to complete the new merged node: The entries in the first node being merged, followed by the splitting key value that is `pulled down' from the parent, followed by the entries in the second non-leaf node gives us a total of 2d keys and 2d + 1 pointers, which is a full non-leaf node. Notice how the splitting key value in the parent is `pulled down,' in contrast to the case of merging two leaf nodes.

Together, the non-leaf node and the sibling to be merged contain only three entries, and they have a total of five pointers to leaf nodes. To merge the two nodes, we also need to `pull down' the index entry in their parent that currently discriminates between these nodes. This index entry has key value 17, and so we create a new entry h17, left-most child pointer in sibling. Now we have a total of four entries and five child pointers, which can fit on one page in a tree of order d=2. Notice that pulling down the splitting key 17 means that it will no longer appear in the parent node following the merge. After we merge the affected non-leaf node and its sibling by putting all the entries on one page and discarding the empty sibling page, the new node is the only child of the old root, which can therefore be discarded. The tree after completing all these steps in the deletion of entry 24* is shown in Figure 9.

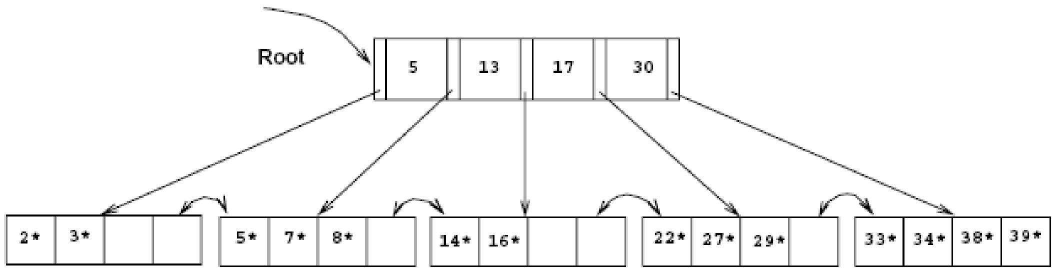


Figure 9 B+ Tree after Deleting Entry 24*

The remaining case is that of redistribution of entries between non-leaf-level pages. To understand this case, consider the intermediate right subtree shown in Figure 8. We would arrive at the same intermediate right subtree if we try to delete 24* from a tree similar to the one shown in Figure 7 but with the left subtree and root key value as shown in Figure 9.20. The tree in Figure 9.20 illustrates an intermediate stage during the deletion of 24*.

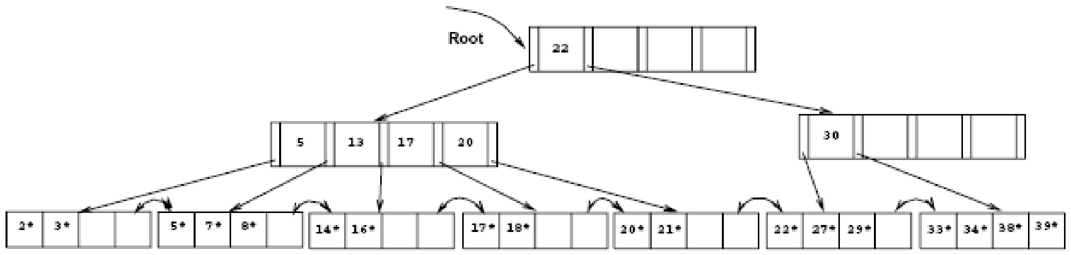


Figure 10 A B+ Tree during a Deletion

In contrast to the case when we deleted 24* from the tree of Figure 9.17, the non-leaf level node containing key value 30 now has a sibling that can spare entries (the entries with key values 17 and 20). We move these entries2 over from the sibling. Notice that in doing so, we essentially `push' them through the splitting entry in their parent node (the root), which takes care of the fact that 17 becomes the new low key value on the right and therefore must replace the old splitting key in the root (the key value 22).

The tree with all these changes is shown in Figure 11.

In concluding our discussion of deletion, we note that we retrieve only one sibling of a node. If this node has spare entries, we use redistribution; otherwise, we merge. If the node has a second sibling, it may be worth retrieving that sibling as well to check for the possibility of redistribution. Chances are high that redistribution will be possible, and unlike merging, redistribution is guaranteed to propagate no further than the parent node. Also, the pages have more space on them, which reduces the likelihood of a split on subsequent insertions. However, the number of times that this case arises is not very high, so it is not essential to implement this refinement of the basic algorithm that we have presented.

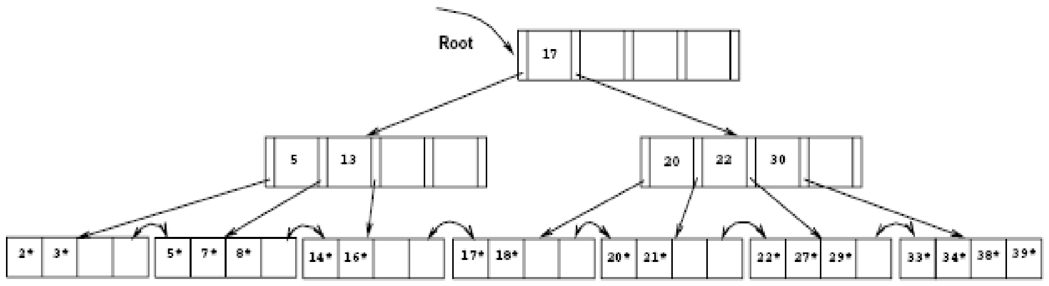


Figure 11 B+ Tree after Deletion