Unit 7

Storage Management and Indexing

File

- ✓ A collection of data or information that has a name, called the filename. Almost all information stored in a computer must be in a file. There are many different types of files: data files, text files, program files, directory files, and so on.
- ✓ The physical or internal level of organization of a database system is concerned with the efficient storage of information in the secondary storage device.
- ✓ The basic problem in the physical database representation is to select suitable file system to store the desired information.

File organization

- ✓ A database is stored as collection of files. Each file is stored as a sequence of records. A record is a sequence of fields. Records are stored on disk blocks.
- ✓ A file organization essentially means organization of records in the file.
- ✓ It is a logical relationship among various records. This method defines how file records are mapped to the disk blocks.

Purpose of file organization

- ✓ File organization makes it easier & faster to perform operations (such as read, write, update, delete etc.) on data stored in files.
- ✓ **Removes data redundancy:** File organization make sure that the redundant and duplicate data gets removed. This alone saves the database from insert, update, delete operation errors which usually happen when duplicate data is present in database.
- ✓ **Save storage cost**: By organizing the data, the redundant data gets removed, which lowers the storage space required to store the data.
- ✓ **Improves accuracy:** When redundant data gets removed and the data is stored in efficient manner, the chances of data gets wrong and corrupted will be minimized.

A file can have mainly two types of records

1. Fixed Length Records
2. Variable Length Records

1. Fixed length records

Consider a records of file deposit of the form

type account=record
account_number:char(10);
branch_name:char(22);
balance:real;
end

If we assume that each character occupies one byte and a real 8 bytes, our deposit record is 40 bytes long. The simplest approach is to use the first 40 bytes for the first record, the next 40 bytes for the second, and so on.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

However, there are two problems with this approach.

1. Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

   Solution: Allocate only as many as records to a block as would fit entirely in the block. This can be computed by
   Number of records in particular block=block size/record size
   And discarding the fractional part. Any remaining bytes of each block are left unused.

2. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

When the record is deleted, we could move the record that came after it into the space formerly occupied by the deleted record and so on, until every record following the deleted record has been moved ahead. Such approach requires moving large number of records.

Record 2 deleted and all records moved

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
~~record 2~~	~~A-215~~	~~Mianus~~	~~700~~
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

It might be easier simply to move the final record of the file into the space occupied by the deleted record.

Record 2 deleted and final record moved

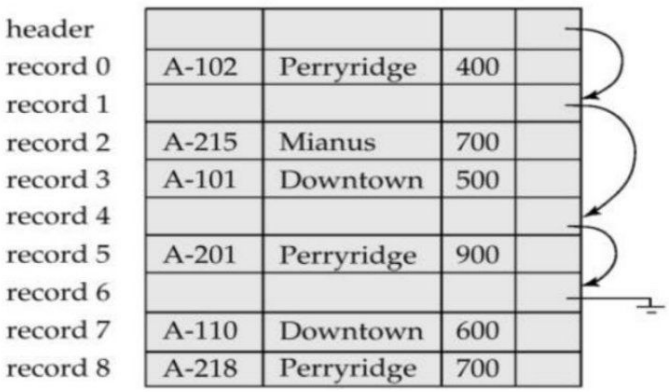
record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

Thus, to avoid above problems we need to introduce an additional structure,

At the beginning of the file, we allocate a certain number of bytes as the file header. The header will contain variety of information about the file. For now, all we need to store there is the address of first record whose contents are deleted.

We use this first record to store the address of second available record and so on. We can think of these stored address as pointers, since they point to the location of a record. Thus deleted record thus form a linked list, which is often referred to as free list. The figure below the file of instructor, with the free list, after records 1,4 and 6 have been deleted.



On insertion of a new record, we use the record pointed to by the header. We change the header pointer to point to the next available record. If no space is available we add the new records to the end of the file.

Insertion and deletion for files of fixed-length records are simple because the space the space made available by a deleted record is exactly the spaced needed to insert the record.

2. Variable–Length records

Variable-length records arise in database systems in several ways:

- ✓ Storage of multiple record types in a file.
- ✓ Record types that allow variable lengths for one or more fields such as strings (varchar)
- ✓ Record types that allow repeating fields ,such as arrays.

One example with variable -length records

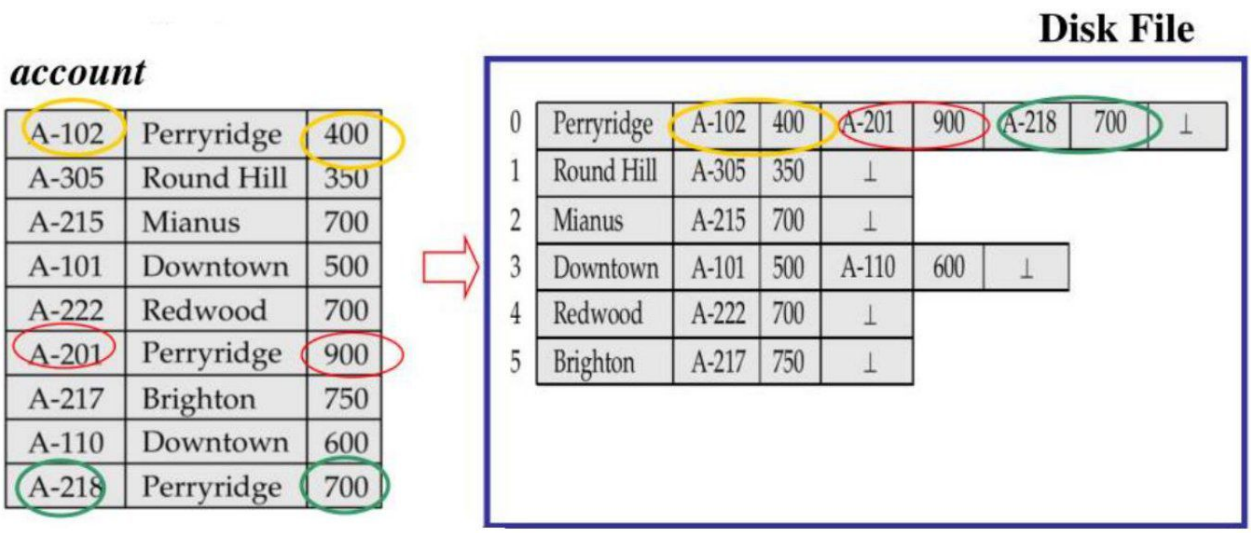
```
type account_list=record
branch_name:char(22);
account_info: array[1.....∞] of record;
                    account_number:char(10)
                    balance:real;
                    end
end
```

In this we define a account_info as an array with an arbitrary number of elements. i.e. the type definition does not limit the number of elements in the array.

Representation Methods

i) Byte String Representation

In byte string representation we attach a special end of record (⊥)



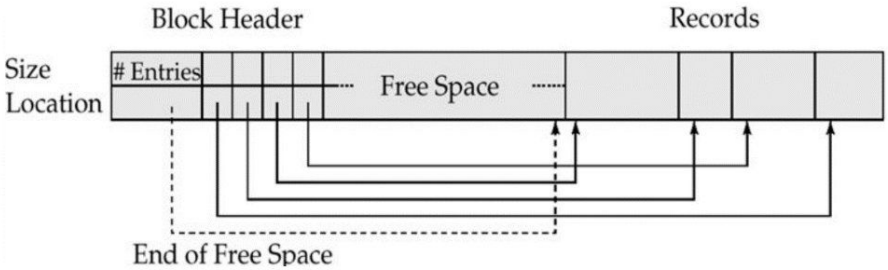
This approach has several disadvantages:

- ✓ It is not easy to reuse the space left by a deleted record
- ✓ In general there is no space for records to grow longer. If the records become longer it must be removed.

A modified form of byte–string representation called slotted page structure.

The slotted page structure there is a header at the beginning of each block, containing the following information.

- ✓ The number of record entries in the header
- ✓ The end of free space in the block
- ✓ An array whose entries contain the location and size of each record



- ✓ The actual records are allocated contiguously in the block, starting from the end of the block.
- ✓ The free space in the block is contiguous, between the final entry in the header array, and the first record.
- ✓ If the records is inserted, space is allocated for it at the end of the free space and the entry containing its size and location is added to the header.
- ✓ If the record is deleted, the space is that it occupies is freed and its entry is set to deleted(*its size is set to -1,for example)*. Further the records in the block before the deleted records are moved, so that free space created by the deletion gets occupied, and all free space is again between the final entry in the array and the first record.
- ✓ Pointers should not point directly to record-instead they should point to the entry for the record in the header.

ii) Fixed –Length representation

- ✓ Use one or more fixed length records

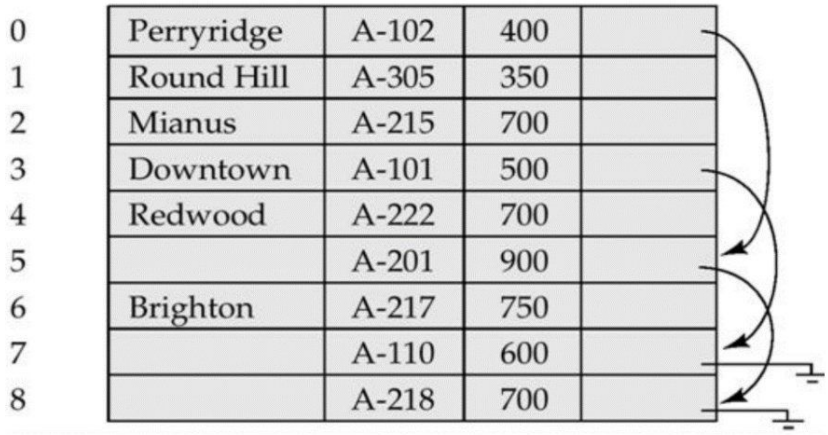
❖ **Reserved space**

- ✓ can use fixed-length records of a known maximum length
- ✓ unused space in shorter records filled with null or end of record symbol

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

❖ **List representation by pointers**

- ✓ A variable-length record is represented by a list of fixed-length records, chained together via pointers.
- ✓ Can used even if the maximum record length is not known

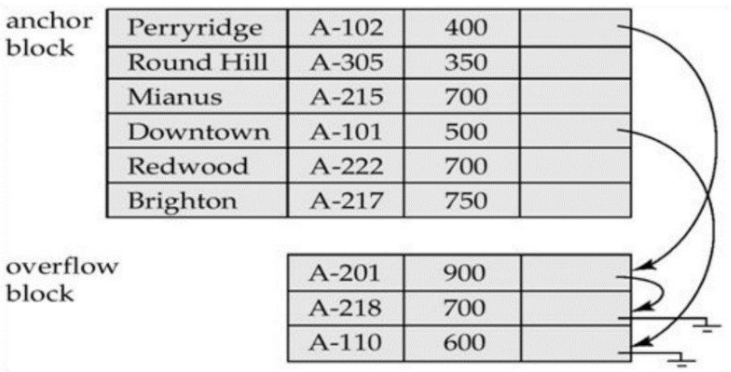


0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

Disadvantage to pointer structure: Space is wasted in all records except the first in chain

Solution is to allow two kinds of block in a file:

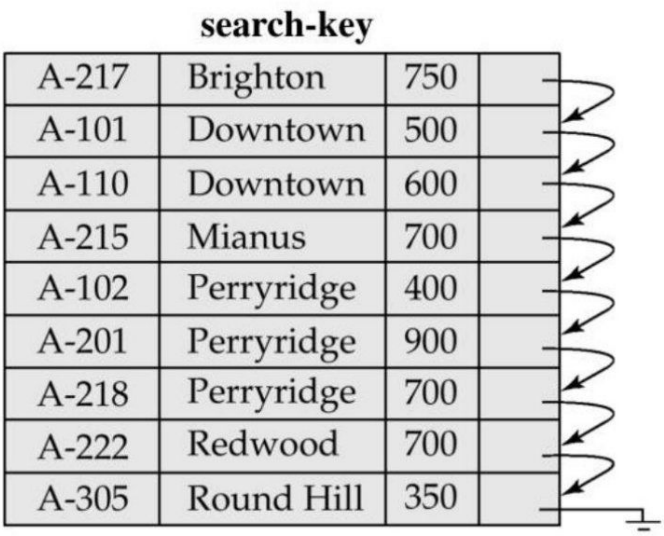
- ✓ **Anchor block-** contains the first records of chain
- ✓ **Overflow block**-contains records other than those that are the first records of chains.



Sequential file organization

- ✓ A sequential file is designed for efficient processing of records in sorted order based on some search key.
- ✓ A search key is any attribute or set of attributes; it need not be the primary key, or even a superkey.
- ✓ To permit fast retrieval of records in search-key order, we chain together records by pointers.
- ✓ The pointer in each record points to the next record in search-key order.
- ✓ Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

In below example, the records are stored in search-key order, using **branch name** as the search key

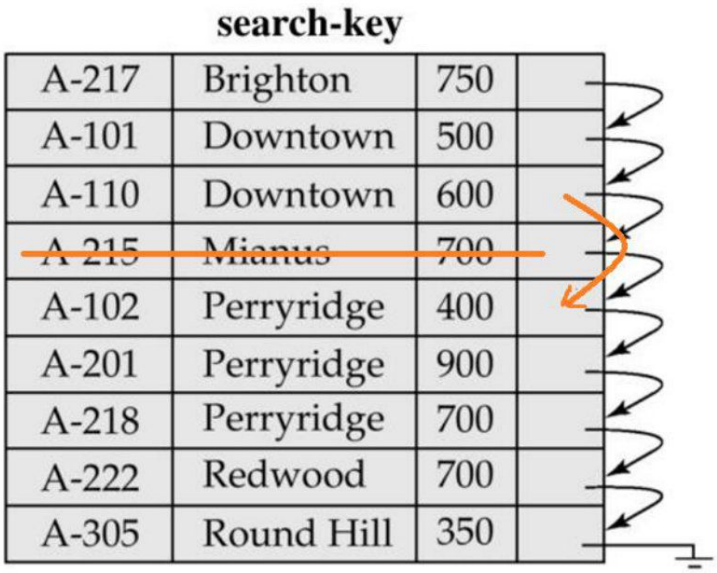


For **insertion** follow the following rule:-

Locate the position where the record is to be inserted

- ✓ if there is free space insert there
- ✓ if no free space, insert the record in an overflow block
- ✓ In either case, pointer chain must be updated

For deletion – use pointer chains



- ✓ Need to reorganize the file from time to time to restore sequential order

Advantages of sequential file organization

- ✓ The sequential file organization is efficient and process faster for the large volume of data.
- ✓ It is simple in design. It requires no much effort to store the data
- ✓ This method can be implemented using cheaper storage devices such as magnetic tapes.
- ✓ It requires fewer efforts to store and maintain data elements.
- ✓ The sequential file organization technique is useful for report generation and statistical computation process.

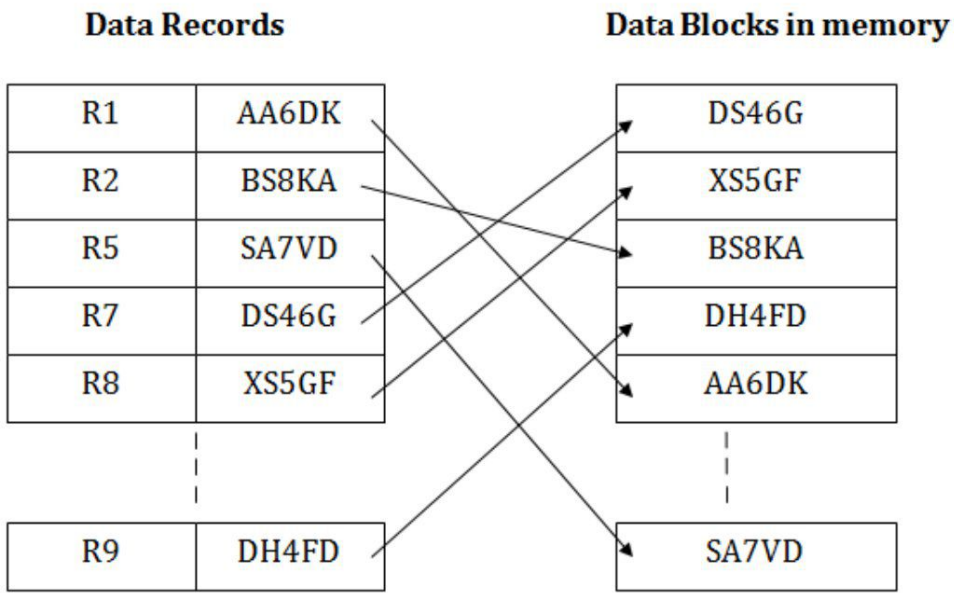
Disadvantages of sequential file organization

- ✓ It will waste time as we cannot jump on a particular record that is required but we have to move sequentially which takes our time.
- ✓ Sorted file method takes more time and space for sorting the records.

Indexed Sequential Access Method (ISAM)

ISAM (Indexed sequential access method) is an advanced sequential file organization method. In this case, records are stored in the file with the help of the primary key. For each primary key, an index value is created and mapped to the record. This index contains the address of the record in the file.

If a record has to be obtained based on its index value, the data block's address is retrieved, and the record is retrieved from memory.



Advantages of ISAM:

- ✓ In this method, each record has the address of its data block, searching a record in a huge database is quick and easy.
- ✓ This method supports range retrieval and partial retrieval of records. Since the index is based on the primary key values, we can retrieve the data for the given range of value. In the same way, the partial value can also be easily searched, i.e., the student name starting with 'JA' can be easily searched.

Disadvantages of ISAM

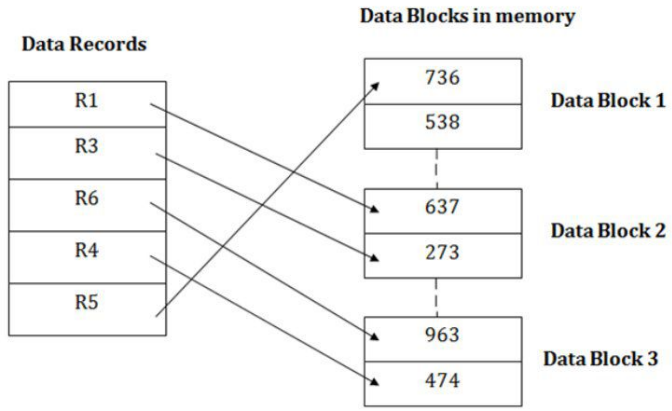
- ✓ This method requires extra space in the disk to store the index value.
- ✓ When the new records are inserted, then these files have to be reconstructed to maintain the sequence.
- ✓ When the record is deleted, then the space used by it needs to be released. Otherwise, the performance of the database will slow down.

Heap file organization

It is the simplest and most basic type of organization. It works with data blocks. In heap file organization, the records are inserted at the file's end. When the records are inserted, it doesn't require the sorting and ordering of records.

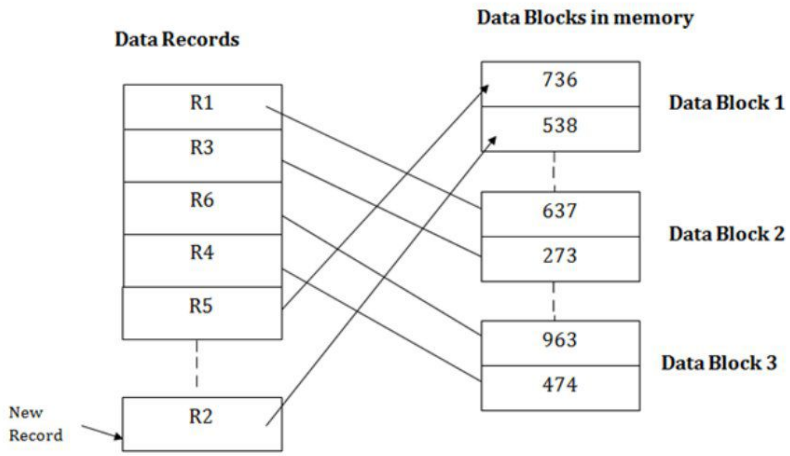
When the data block is full, the new record is stored in some other block. This new data block need not to be the very next data block, but it can select any data block in the memory to store new records. The heap file is also known as an unordered file.

In the file, every record has a unique id, and every page in a file is of the same size. It is the DBMS responsibility to store and manage the new records.



Insertion of a new record

Suppose we have five records R1, R3, R6, R4 and R5 in a heap and suppose we want to insert a new record R2 in a heap. If the data block 3 is full then it will be inserted in any of the database selected by the DBMS, let's say data block 1.



If we want to search, update or delete the data in heap file organization, then we need to traverse the data from staring of the file till we get the requested record.

If the database is very large then searching, updating or deleting of record will be time-consuming because there is no sorting or ordering of records. In the heap file organization, we need to check all the data until we get the requested record.

Advantages of Heap file organization

- ✓ It is a very good method of file organization for bulk insertion. If there is a large number of data which needs to load into the database at a time, then this method is best suited.
- ✓ In case of a small database, fetching and retrieving of records is faster than the sequential record.

Disadvantages of Heap file organization

- ✓ This method is inefficient for the large database because it takes time to search or modify the record.
- ✓ This method is inefficient for large databases.

Indexing

- ✓ Indexing mechanisms used to speed up access to desired data.(E.g., author catalog in library)
- ✓ An index takes as input a search key value and returns the address of the records hold that values.
- ✓ **Search Key** - attribute to set of attributes used to look up records in a file.
- ✓ An **index file** consists of records (called **index entries**) of the form

Search key	Pointer

- ✓ Index files are typically much smaller than the original file

Two basic kinds of indices:

- **Ordered indices:** search keys are stored in sorted order
- **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function"

Index evaluation metrics:

1. **Access types:** The type of access that are supported efficiently. Access types can include finding records with a specified attribute value and finding records whose attribute values fall in a specified range.
2. **Access time:** The time it takes to find a particular data item, or set of items
3. **Insertion time:** The time it takes to insert a new data item
4. **Deletion time:** The time it takes to delete a data item. This value includes the time it takes to find the item to be deleted, as well as the time it takes to update the index structure.
5. **Space overhead:** Additional space occupied by an index structure

Ordered Indices

In an **ordered index,** index entries are stored sorted on the search key value.

Primary index: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.

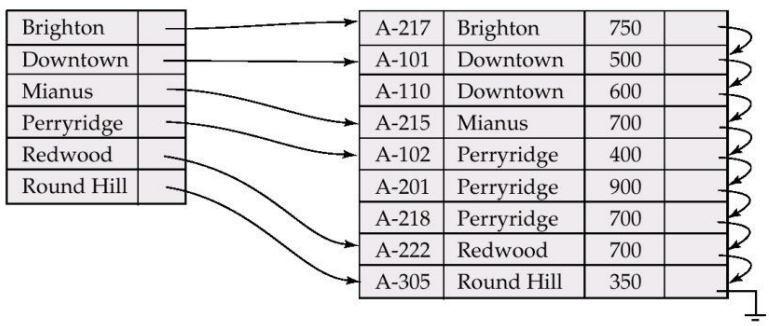
- ✓ Also called **clustering index**
- ✓ The search key of a primary index is usually but not necessarily the primary key.

Types

1. Dense index
2. Sparse index

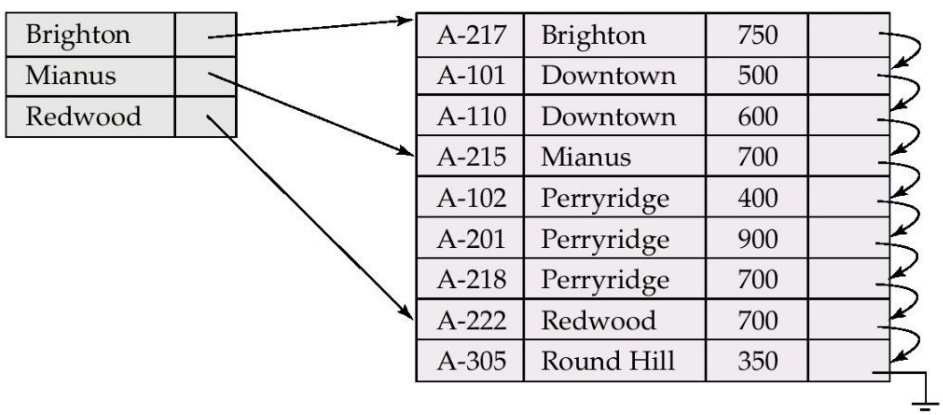
1. Dense Index

- ✓ An index record appears for every search- key value in a file.
- ✓ Index record contain 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 .
- ✓ It require more memory space for index. Records are accessed fast.



2. Sparse Index

- ✓ Contains index records for only some search-key values.
- ✓ Sparse index can be used only if the relation the relation is stored in sorted order of the search key.
- ✓ To locate a record, we find the index 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 that record pointed to by the index record, and proceed along the pointer in file (i.e. sequentially) until we find the desired record.



Comparison with dense index

- ✓ Less space and less maintenance overhead for insertions and deletions
- ✓ Generally slower than dense index for locating records.

Secondary index

- An index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**
- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index satisfy some condition.
  - Example 1: In the *account* database stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value; index record points to a bucket that contains pointers to all the actual records with that particular search-key value.

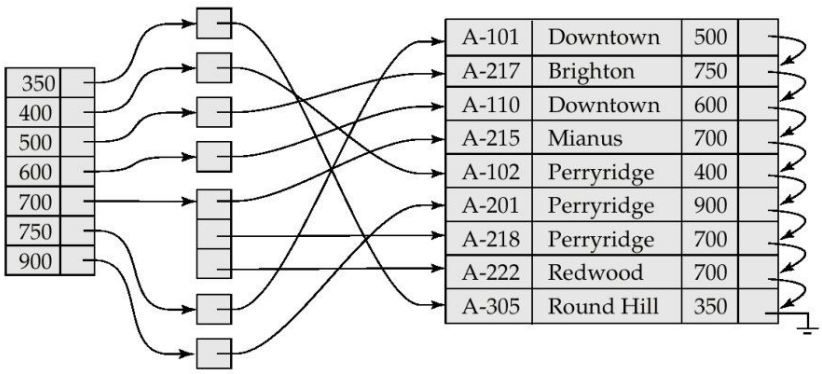


Figure: Secondary Index on balance field of account

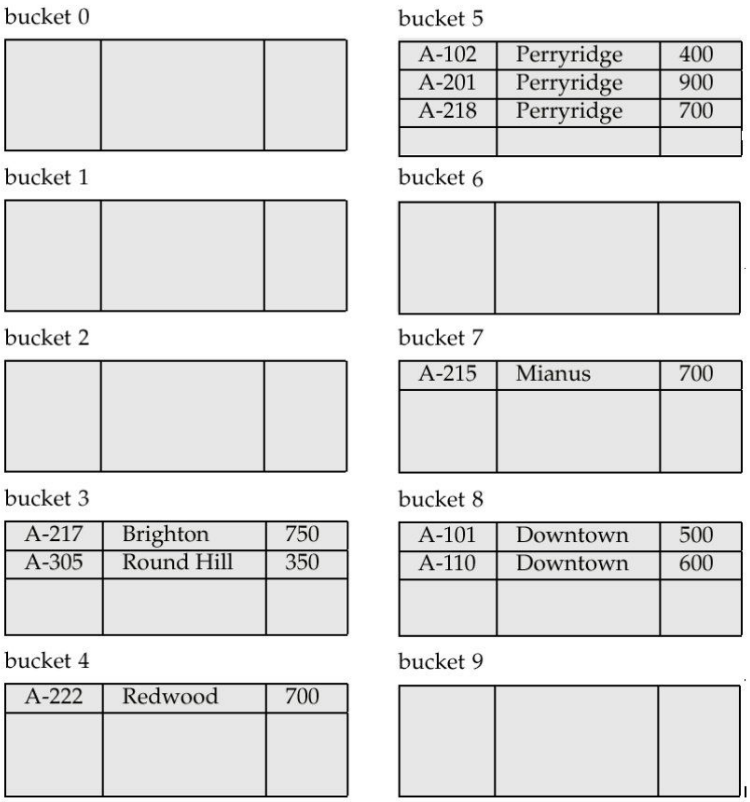
Hash file organization

- ✓ A bucket is a unit of storage containing one or more records(a bucket is typically a disk block)
- ✓ In a hash file organization we obtain the block directly from its search key value using a hash function.
- ✓ Hash function *h* is a function from the set of all search-key values *K* to the set of all bucket addresses *B.*
- ✓ Hash function is used to locate records for access, insertion as well as deletion.
- ✓ Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record

Hash file organization of *account* file, using *branch-name* as key

- ✓ There are 10 buckets,
- ✓ The binary representation of the *i*th character is assumed to be the integer *i.*
- ✓ The hash function returns the sum of the binary representations of the characters modulo 10

E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3



Bucket overflow

We have assumed that, when a record is inserted, the bucket to which it is mapped has space to store the record. If the bucket does not have enough space a bucket overflow is said to occur. Bucket overflow can occur for several reasons:

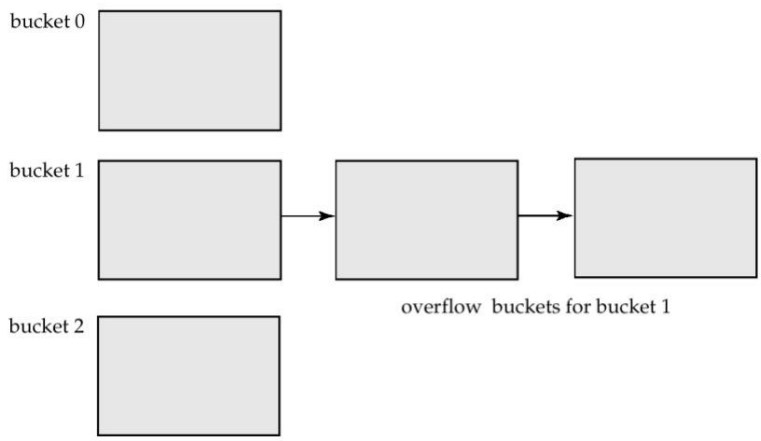
- **Insufficient buckets**
- ✓ Number of buckets (n_B) must be chosen such that $n_B > n_r/f_r$,where n_r denotes the total number of records that will be stored and f_r denotes the total number of records that will fit in a bucket.
- **Skew**
- ✓ Some buckets are assigned to more records than others, so a bucket may overflow even when other buckets still have space. This situation is called bucket skew. Skew can occur for two reasons:

1. Multiple records may have the same search key
2. The chosen hash function may result in non-uniform distribution of search keys.

The worst possible hash function maps all search key values to same bucket. Such a hash function is undesirable because all the records have to be kept in same bucket. An ideal hash function distributes the stored key uniformly across the buckets so that every bucket has the same number of records.

Handling Bucket overflow

We handle bucket overflow by using overflow buckets. If a record must be inserted in bucket b and bucket b is already full, the system provides overflow bucket for b and insert the record into the overflow bucket as shown in above figure. If the overflow bucket is also full then the system provides another overflow bucket and so on. All the overflow buckets of the given bucket are chained together in a linked list. Overflow handling using such linked list is called overflow chaining.

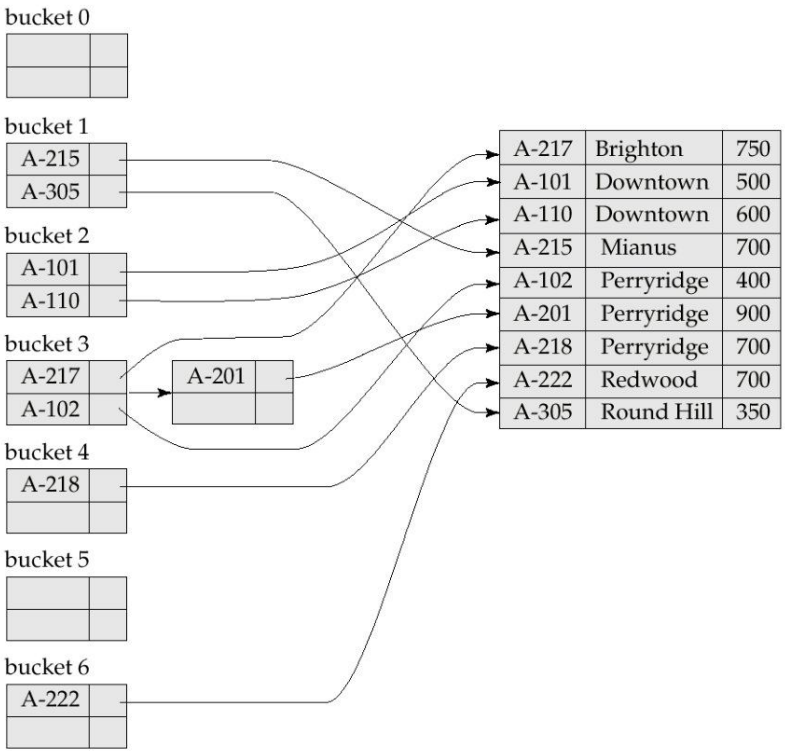


Above scheme is called closed hashing**.**

Alternative approach called open hashing. The set of buckets is fixed and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other buckets in the initial set of buckets B. One policy is to use the next bucket that has space; this policy is called linear probing.

Hash index

- ✓ A hash index organizes the search keys with their associated pointers into a hash file structure.
- ✓ We apply a hash function on a search key to identify a bucket, and store the key and its associated pointers in the bucket (or in overflow buckets).
- ✓ A hash function, h(K), is a mapping function that maps all the set of search-keys K to the bucket where keys and its associated pointers are placed.



- ✓ The hash function in the above function is H(K)= Sum of the digits of account number % 7
- ✓ The hash index has 7 buckets each size 2. One of the bucket has 3 keys mapped so it has an overflow bucket.
- ✓ Formally let K denote the set of all search keys value, B denote the set of all bucket address and h denote the hash function. To insert a record with search key ki , we compute h(Ki) which gives the address of bucket for that record. That record is then stored in that bucket.
- ✓ Deletion is also simple. If the search key value of the record to be deleted is Ki. We compute h(ki) to find the corresponding bucket for that record and delete the record from the bucket.

B+ tree Index files

A B+ Tree is simply a balanced binary search tree, in which all data is stored in the leaf nodes, while the internal nodes store just the indices.

The main disadvantage of indexed-sequential files

- ➢ Performance degrades as file grows, since many overflow blocks get created.
- ➢ Periodic reorganization of entire file is required.

B+ tree file structure maintains the efficient tree as it automatically reorganizes itself after every insertion and deletion.

Properties of B+ Trees

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf (Internal node) has between [n/2] and n children.
- A leaf node has between [(n–1)/2] and n–1 value

Special cases:

- ✓ If the root is not a leaf, it has at least 2 children.
- ✓ If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n-1) values.

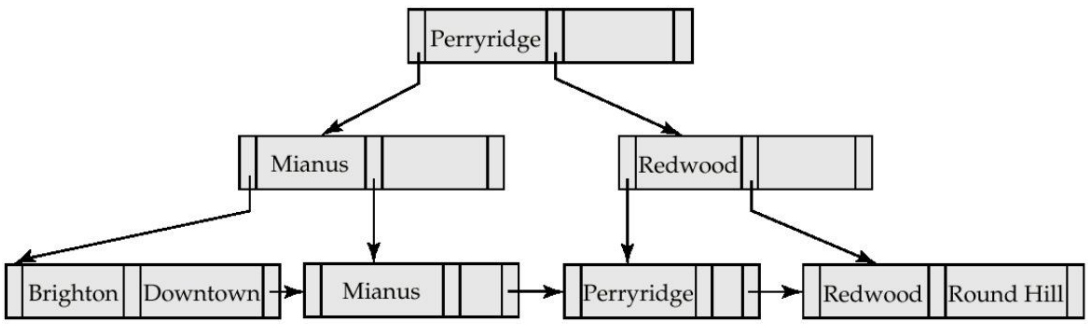
B+ tree node structure

- ➢ K_i are the search-key values
- ➢ P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).

The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \ldots < K_{n-1}$$

Example of B+ tree of order(n=3)



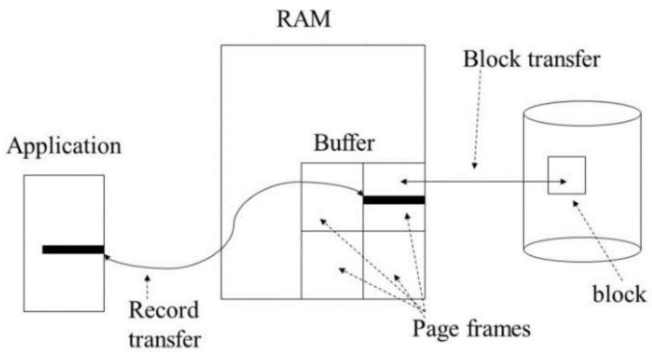
Advantages of B+ tree

- ✓ Since all records are stored only in the leaf node and are sorted sequential linked list, searching is becomes very easy.
- ✓ Using B+, we can retrieve range retrieval or partial retrieval. Traversing through the tree structure makes this easier and quicker.
- ✓ As the number of record increases/decreases, B+ tree structure grows/shrinks. There is no restriction on B+ tree size, like we have in ISAM.
- ✓ Since it is a balance tree structure, any insert/ delete/ update does not affect the performance.
- ✓ Since we have all the data stored in the leaf nodes and more branching of internal nodes makes height of the tree shorter. This reduces disk I/O. Hence it works well in secondary storage devices.

Buffer management in DBMS

Database buffer

- ✓ A database buffer is a section in the main memory that is used for the temporary storage of copies of disk block.
- ✓ Goal is to minimize the number of transfers between the disk storage and the main memory (RAM).



Buffer Manger

- ✓ A Buffer Manager is responsible for allocating space to the buffer in order to store data into the buffer.

When user request a particular block then

1. If block is already in the buffer
   - ➢ The buffer manager provides the block address in the main memory.
2. If the block is not available in the buffer
   - ➢ The buffer manager allocates the space in the buffer for a block.
   - ➢ If free space is not available, it throws out some existing blocks from the buffer to allocate the required space for the new block.
   - ➢ The blocks which are thrown are written back to the disk only if they are recently modified when writing on the disk.
   - ➢ Once the space is allocated in buffer, the buffer manager reads the requested block from the disk to the buffer and then passes the address of the requested block to the user in the main memory.

Buffer Replacement Strategy:

- ✓ If no space is left in the buffer, it is required to remove an existing block from the buffer before allocating the new one.
- ✓ The various operating system uses the LRU (least recently used) scheme.
- ✓ In LRU, the block that was least recently used is removed from the buffer and written back to the disk.
- ✓ Such type of replacement strategy is known as Buffer Replacement Strategy.
- ✓ Other buffer replacement strategy can MRU(Most Recently Used),FIFO (First In First Out) etc.

Spanned and unspanned records

- ✓ The records of a file must be allocated to disk blocks because a block is the unit of data transfer between disk and memory.
- ✓ When the block size is larger than the record size, each block will contain numerous records, although some of the files may have unusually large records that cannot fit in one block.

  Blocking factor (bfr) - the number of records that can fit into a single block

  bfr= B/R

  Where,

  - ➢ B=Block size in bytes
  - ➢ R=Record size in bytes

  Example:

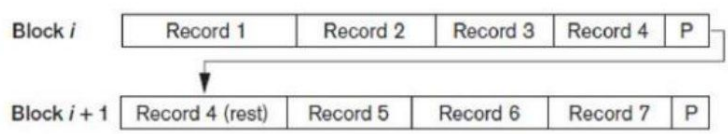
  Block size B=2000 bytes
  Record size R=100 bytes

  Now, blocking factor (bfr)= **2000/100=20**

In general, R may not divide B exactly, so we have some unused space in each block equal to

B – (bfr * R) bytes.

To utilize this unused space, we can store part of a record on one block and the rest on another. A pointer at the end of the first block points to the block containing the remainder of the record. This organization is called **spanned.**



When extra space in the block is left unused the record organization is called **unspanned.**