CHAPTER - 8

File organization and Indexing

8.1 DBMS Storage System

Databases are stored in file formats, which contains records. At physical level, actual data is stored in electromagnetic format on some device capable of storing it for a longer amount of time. Several types of data storage exist in most computer systems. Among the media typically available are these:

- **Cache.** The cache is the fastest and most costly form of storage. Cache memory is small; its use is managed by the computer system hardware. We shall not be concerned about managing cache storage in the database system.
- **Main memory.** The storage medium used for data that are available to be operated on is main memory. The general-purpose machine instructions operate on main memory. Although main memory may contain many megabytes of data, or even gigabytes of data in large server systems, it is generally too small (or too expensive) for storing the entire database. The contents of main memory are usually lost if a power failure or system crash occurs.
- **Flash memory.** Also known as electrically erasable programmable read-only memory (EEPROM), flash memory differs from main memory in that data survive power failure. Reading data from flash memory takes less than 100 nanoseconds (a nanosecond is 1/1000 of a microsecond), which is roughly as fast as reading data from main memory.
- **Magnetic-disk storage.** The primary medium for the long-term on-line storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The system must move the data from disk to main memory, so that they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.
- **Optical storage.** The most popular forms of optical storage are the compact disks (CD), which can hold about 640 megabytes of data, and the digital video disk (DVD) which can hold 4.7 or 8.5 gigabytes of data per side of the disk (or up to 17 gigabytes on a two-sided disk). Data are stored optically on a disk, and are read by a laser. The optical disks used in read-only compact disks (CD-ROM) or read-only digital video disk (DVD-ROM) cannot be written, but are supplied with data prerecorded.
- **Tape storage.** Tape storage is used primarily for backup and archival data. Although magnetic tape is much cheaper than disks, access to data is much slower, because the tape must be accessed sequentially from the beginning. For this reason, tape storage is referred to as **sequential-access** storage. In contrast, disk storage is referred to as **direct-access** storage because it is possible to read data from any location on disk.

The Memory Hierarchy

A computer system has well-defined hierarchy of memory. CPU has inbuilt registers, which saves data being operated on. Computer system has main memory, which is also directly accessible by CPU. Because the access time of main memory and CPU speed varies a lot, to minimize the loss cache memory is introduced. Cache memory contains most recently used data and data which may be referred by CPU in near future.

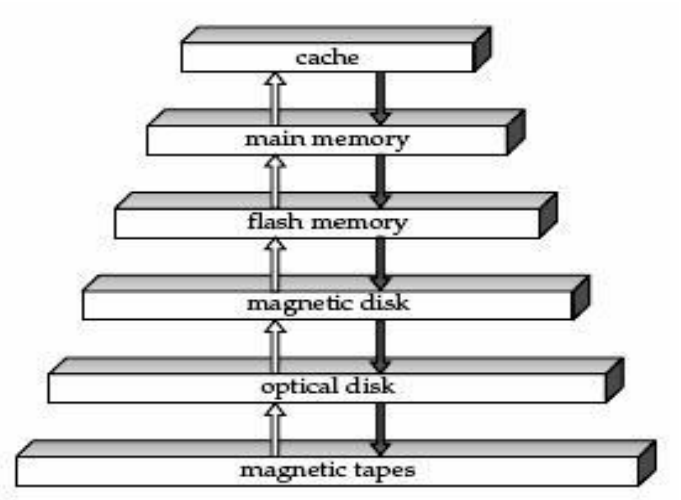
The design constraints on a computer's memory can be summed up by cost, capacity and access time. A variety of technologies are used to implement memory systems, and across this spectrum of technologies, the following relationships hold:

* Faster access time, greater cost per bit
* Greater capacity, smaller cost per bit
* Greater capacity, slower access time

A typical hierarchy is illustrated in following figure. As one goes down the hierarchy, the following occur:

- Decreasing cost per bit
- Increasing capacity & access time
- Decreasing frequency of access of the memory by the process

Hierarchy List: Registers > L1 Cache > L2 Cache > Main memory > Disk cache > Disk > Optical > Tape



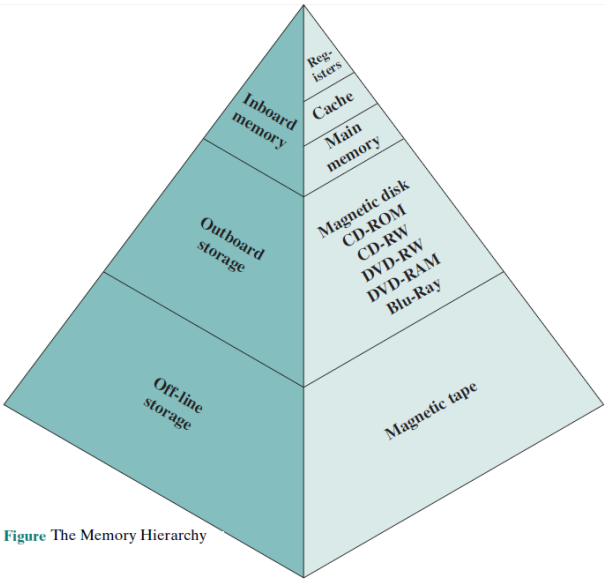


Figure The Memory Hierarchy

The fastest storage media –for example, cache and main memory –are referred to as **primary storage**. The media in the next level in the hierarchy–for example, magnetic disks –are referred to as **secondary storage**, or **online storage**. The media in the lowest level in the hierarchy – for example, magnetic tape and optical-disk jukeboxes –are referred to as **tertiary storage**, or **offline storage**.

In addition to the speed and cost of the various storage systems, there is also the issue of storage volatility. **Volatile storage** loses its contents when the power to the device is removed. In the hierarchy shown in Figure 4.1, the storage systems from main memory up are volatile, whereas the storage systems below main memory are nonvolatile. In the absence of expensive battery and generator backup systems, data must be written to **nonvolatile storage** for safe keeping.

Magnetic Disks

The primary medium for the long-term on-line storage of data is the magnetic disk. Usually, the entire database is stored on magnetic disk. The system must move the data from disk to main memory, so that they can be accessed. After the system has performed the designated operations, the data that have been modified must be written to disk.

Hard disk drives are the most common secondary storage devices in present day computer systems. These are called magnetic disks because it uses the concept of magnetization to store information. Hard disks consist of metal disks coated with magnetizable material. These disks are placed vertically a spindle. A read/write head moves in between the disks and is used to magnetize or de-magnetize the spot under it. Magnetized spot can be recognized as 0 (zero) or 1 (one).

Hard disks are formatted in a well-defined order to stored data efficiently. A hard disk plate has many concentric circles on it, called tracks. Every track is further divided into sectors. A sector on a hard disk typically stores 512 bytes of data.

Magnetic disks provide the bulk of secondary storage for modern computer systems. Disk capacities have been growing at over 50 percent per year, but the storage requirements of large applications have also been growing very fast, in some cases even faster than the growth rate of disk capacities. A large database may require hundreds of disks.

Physical Characteristics of Disks

Physically, disks are relatively simple. Each disk **platter** has a flat circular shape. Its two surfaces are covered with a magnetic material, and information is recorded on the surfaces. Platters are made from rigid metal or glass and are covered (usually on both sides) with magnetic recording material. We call such magnetic disks **hard disks**, to distinguish them from **floppy disks**, which are made from flexible material.

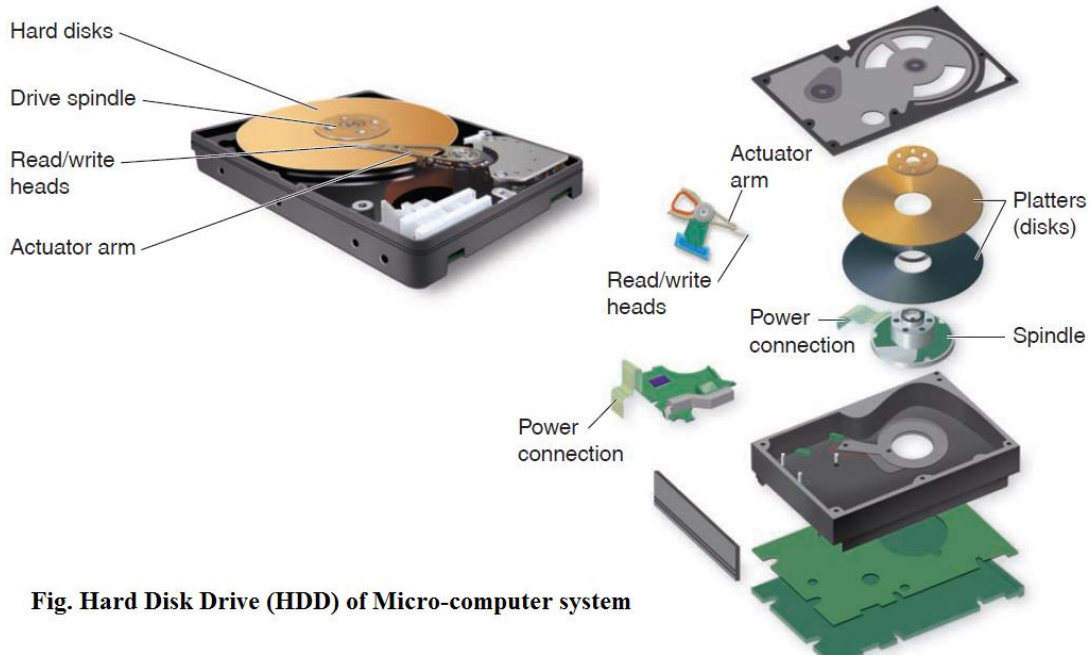


Fig. Hard Disk Drive (HDD) of Micro-computer system

The disk surface is logically divided into **tracks**, which are subdivided into **sectors**. A **sector** is the smallest unit of information that can be read from or written to the disk. The **read–write head** stores information on a sector magnetically as reversals of the direction of magnetization of the magnetic material. There may be hundreds of concentric tracks on a disk surface, containing thousands of sectors. Each side of a platter of a disk has a read–write head, which moves across the platter to access different tracks. A disk typically contains many platters, and the read–write heads of all the tracks are mounted on a single assembly called a **disk arm**, and move together. The disk platters mounted on a spindle and the heads mounted on a disk arm are together known as **head– disk assemblies**.

A **fixed-head disk** has a separate head for each track. This arrangement allows the computer to switch from track to track quickly, without having to move the head assembly, but because of the large number of heads, the device is extremely expensive. Some disk systems have multiple disk arms, allowing more than one track on the same platter to be accessed at a time. Fixed-head disks and multiple-arm disks were used in high-performance mainframe systems, but are no longer in production.

A **disk controller** interfaces between the computer system and the actual hardware of the disk drive. It accepts high-level commands to read or write a sector, and initiates actions, such as moving the disk arm to the right track and actually reading or writing the data. Disk

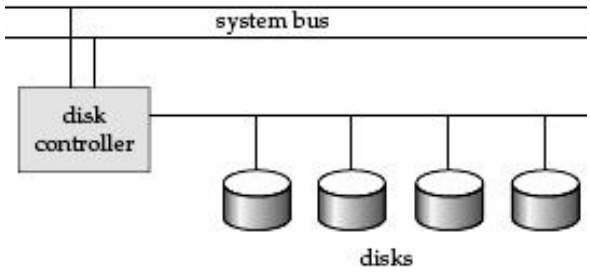


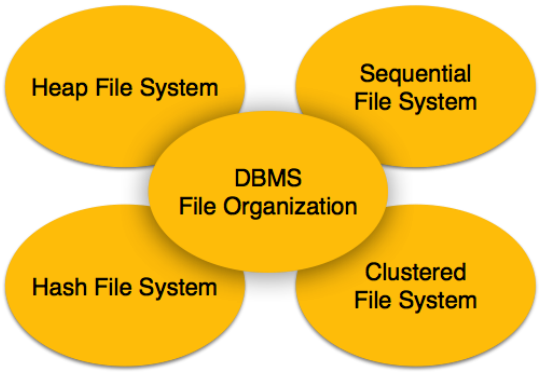
Fig. Disk controller interface

controllers also attach **checksums** to each sector that is written; the checksum is computed from the data written to the sector. When the sector is read back, the controller computes the checksum again from the retrieved data and compares it with the stored checksum; if the data are corrupted, with a high probability the newly computed checksum will not match the stored checksum. If such an error occurs, the controller will retry the read several times; if the error continues to occur, the controller will signal a read failure.

In the **storage area network** (**SAN**) architecture, large numbers of disks are connected by a high-speed network to a number of server computers. The disks are usually organized locally using **redundant arrays of independent disks** (**RAID**) storage organizations, but the RAID organization may be hidden from the server computers: the disk subsystems pretend each RAID system is a very large and very reliable disk.

8.2 File organization

Relative data and information is stored collectively in file formats. A *file* is organized logically as a sequence of records. File size depends on the size of records. The records are of two type: *Fixed-Length Records* and *Variable-Length Records*. These records are mapped onto disk blocks. Files are provided as a basic construct in operating systems, so we shall assume the existence of an underlying *file system*. We need to consider ways of representing logical data models in terms of files.



The method of mapping file records to disk blocks defines file organization, i.e. how the file records are organized. Several of the possible ways of organizing records in files are:

- **Heap File Organization**: When a file is created using Heap File Organization mechanism, the Operating Systems allocates memory area to that file without any further accounting details. File records can be placed anywhere in that memory area. It is the responsibility of software to manage the records. Heap File does not support any ordering, sequencing or indexing on its own.
- **Sequential File Organization**: Every file record contains a data field (attribute) to uniquely identify that record. In sequential file organization mechanism, records are placed in the file in the some sequential order based on the unique key field or search key. Practically, it is not possible to store all the records sequentially in physical form.
- **Hash File Organization**: This mechanism uses a Hash function computation on some field of the records. As we know, that file is a collection of records, which has to be mapped on some block of the disk space allocated to it. This mapping is defined that the hash computation. The output of hash function determines the location of disk block where the records may exist.
- **Clustered File Organization**: Clustered file organization is not considered good for large databases. In this mechanism, related records from one or more relations are kept in a same disk block, that is, the ordering of records is not based on primary key or search key. This organization helps to retrieve data easily based on particular join condition. Other than particular join condition, on which data is stored, all queries become more expensive.

A. 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.

Figure shows a sequential file of *account* records taken from our banking example. In that example, the records are stored in search-key order, using *branchname* as the search key.

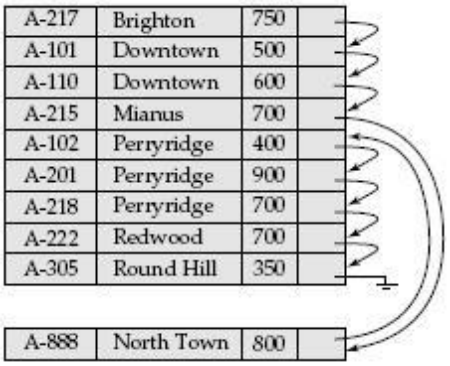
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

The sequential file organization allows records to be read in sorted order; that can be useful for display purposes, as well as for certain query-processing algorithms. It is difficult, however, to maintain physical sequential order as records are inserted and deleted, since it is costly to move many records as a result of a single insertion or deletion. We can manage deletion by using pointer chains, as we saw previously.

For insertion, we apply the following rules:

1. Locate the record in the file that comes before the record to be inserted in search-key order.
2. If there is a free record (that is, space left after a deletion) within the same block as this record, insert the new record there. Otherwise, insert the new record in an *overflow block*. In either case, adjust the pointers so as to chain together the records in search-key order.

Figure shows the file of above figure after the insertion of the record (North Town, A-888, 800). The structure in figure allows fast insertion of new records, but forces sequential file-processing applications to process records in an order that does not match the physical order of the records.



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	
A-888	North Town	800	

B. Indexed Sequential Access Method (ISAM)

We know that information in the DBMS files is stored in form of records. Every record is equipped with some key field, which helps it to be recognized uniquely. Indexing is a data structure technique to efficiently retrieve records from database files based on some attributes on which the indexing has been done. Indexing in database systems is similar to the one we see in books. Indexing is defined based on its indexing attributes. Indexing can be one of the following types:

- **Primary Index:** If index is built on ordering 'key-field' of file it is called Primary Index. Generally it is the primary key of the relation.
- **Secondary Index:** If index is built on non-ordering field of file it is called Secondary Index.
- **Clustering Index:** If index is built on ordering non-key field of file it is called Clustering Index.

Ordering field is the field on which the records of file are ordered. It can be different from primary or candidate key of a file. Ordered Indexing is of two types:

- Dense Index

- Sparse Index

Dense Index

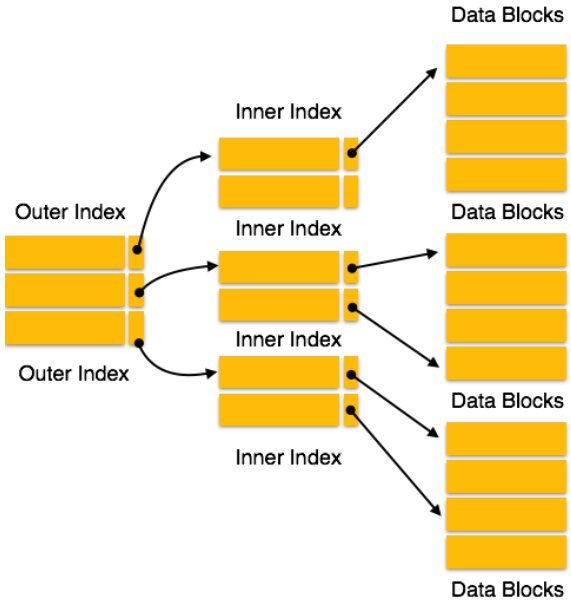
In dense index, there is an index record for every search key value in the database. This makes searching faster but requires more space to store index records itself. Index record contains search key value and a pointer to the actual record on the disk.

Sparse Index

In sparse index, index records are not created for every search key. An index record here contains search key and actual pointer to the data on the disk. To search a record we first proceed by index record and reach at the actual location of the data. If the data we are looking for is not where we directly reach by following index, the system starts sequential search until the desired data is found.

Multilevel Index

Index records are comprised of search-key value and data pointers. This index itself is stored on the disk along with the actual database files. As the size of database grows so does the size of indices. There is an immense need to keep the index records in the main memory so that the search can speed up. If single level index is used then a large size index cannot be kept in memory as whole and this leads to multiple disk accesses.



Multi-level Index helps breaking down the index into several smaller indices in order to make the outer most level so small that it can be saved in single disk block which can easily be accommodated anywhere in the main memory.

Secondary Indices

Secondary indices must be dense, with an index entry for every search-key value, and a pointer to every record in the file. A clustering index may be sparse, storing only some of the search-key values, since it is always possible to find records with intermediate search-key values by a sequential access to a part of the file, as described earlier. If a secondary index stores only some of the search-key values, records with intermediate search-key values may be anywhere in the file and, in general, we cannot find them without searching the entire file.

A secondary index on a candidate key looks just like a dense clustering 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 clustering indices. If the search key of a clustering 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 clustering index, rather than by the search key of the secondary index. Therefore, a secondary index must contain pointers to all the records.

We can use an extra level of indirection to implement secondary indices on search keys that are not candidate keys. The pointers in such a secondary index do not point directly to the file. Instead, each points to a bucket that contains pointers to the file. Figure shows the structure of a secondary index that uses an extra level of indirection on the *instructor* file, on the search key *salary*.

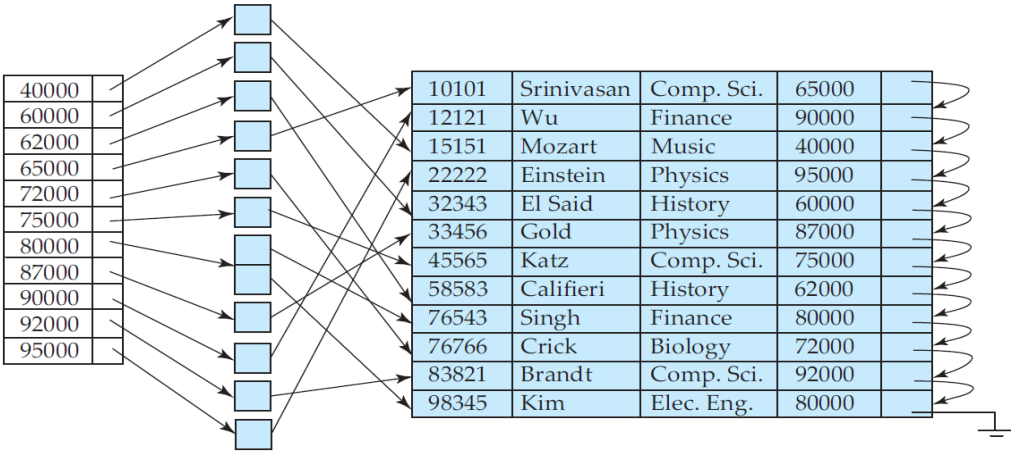
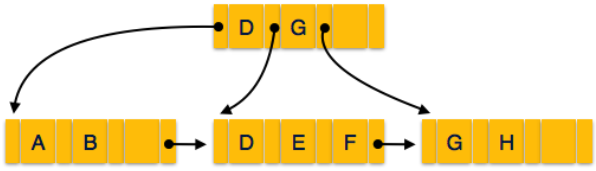


Figure Secondary index on *instructor* file, on noncandidate key *salary*.

C. B^+ Tree

The main disadvantage of the index-sequential file organization is that performance degrades as the file grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the file, frequent reorganizations are undesirable. The **B+-tree** index structure is the most widely used of several index structures that maintain their efficiency despite insertion and deletion of data. A B+-tree index takes the form of a **balanced tree** in which every path from the root of the tree to a leaf of the tree is of the same length. Each non leaf node in the tree has between $[n/2]$ and n children, where n is fixed for a particular tree.

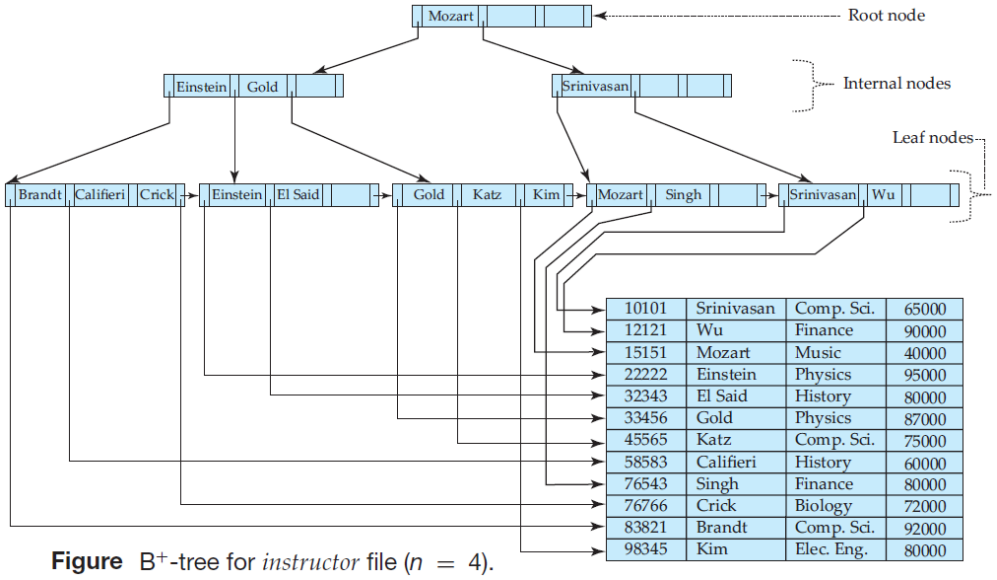
The B+-tree structure imposes performance overhead on insertion and deletion, and adds space overhead. The overhead is acceptable even for frequently modified files, since the cost of file reorganization is avoided. Furthermore, since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space. This space overhead, too, is acceptable given the performance benefits of the B+-tree structure.



B tree is multi-level index format, which is balanced binary search trees. As mentioned earlier single level index records becomes large as the database size grows, which also degrades performance. All leaf nodes of B^+ tree denote actual data pointers. B^+ tree ensures that all leaf nodes remain at the same height, thus balanced. Additionally, all leaf nodes are linked using link list, which makes B^+ tree to support random access as well as sequential access.

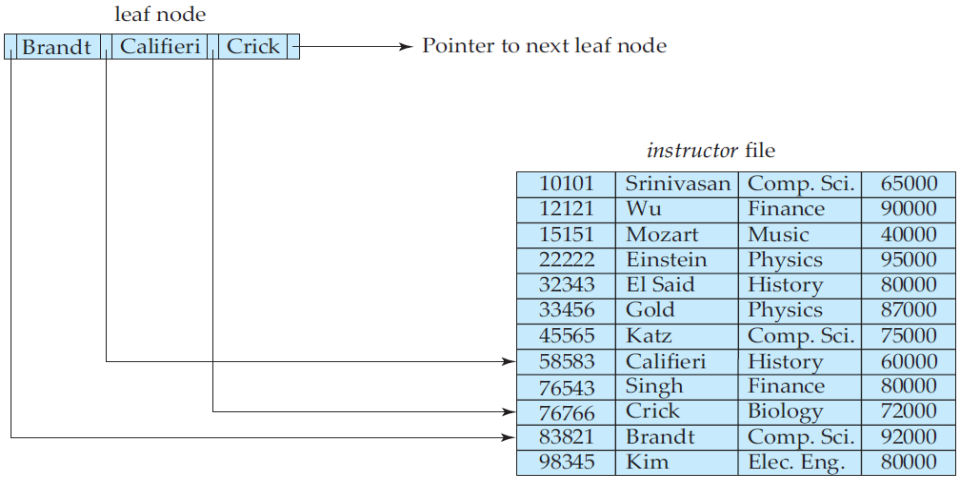
Structure of B⁺ tree

A B+-tree index is a multilevel index, but it has a structure that differs from that of the multilevel index-sequential file. Every leaf node is at equal distance from the root node. A B⁺ tree is of order n where n is fixed for every B⁺ tree.



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

Figure B⁺-tree for *instructor* file ($n = 4$).



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

Figure A leaf node for *instructor* B⁺-tree index ($n = 4$).

Internal nodes:

- Internal (non-leaf) nodes contains at least ⌈n/2⌉ pointers, except the root node.
- At most, internal nodes contain n pointers.

Leaf nodes:

- Leaf nodes contain at least ⌈n/2⌉ record pointers and ⌈n/2⌉ key values
- At most, leaf nodes contain n record pointers and n key values
- Every leaf node contains one block pointer P to point to next leaf node and forms a linked list.

B⁺ tree insertion

- B⁺ tree are filled from bottom. And each node is inserted at leaf node.
- **If leaf node overflows:**
  - Split node into two parts
  - Partition at i = ⌊(m+1)/2⌋
  - First i entries are stored in one node
  - Rest of the entries (i+1 onwards) are moved to a new node
  - i^{th} key is duplicated in the parent of the leaf

- **If non-leaf node overflows:**
  - Split node into two parts
  - Partition the node at i = $\lceil (m+1)/2 \rceil$
  - Entries upto i are kept in one node
  - Rest of the entries are moved to a new node

B^+ tree deletion

- B^+ tree entries are deleted leaf nodes.
- The target entry is searched and deleted.
  - If it is in internal node, delete and replace with the entry from the left position.
- After deletion underflow is tested
  - If underflow occurs
    - Distribute entries from nodes left to it.
  - If distribution from left is not possible
    - Distribute from nodes right to it
  - If distribution from left and right is not possible
    - Merge the node with left and right to it.

D. DBMS Hashing

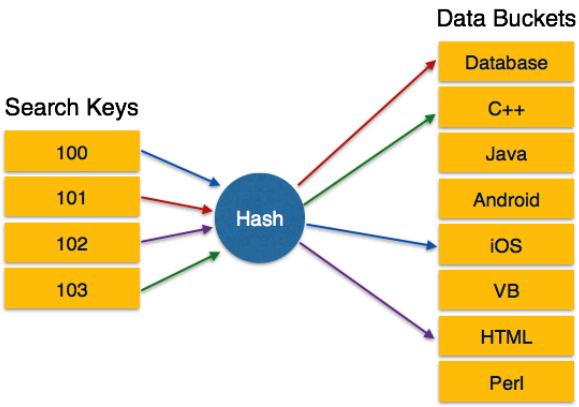
For a huge database structure it is not sometime feasible to search index through all its level and then reach the destination data block to retrieve the desired data. Hashing is an effective technique to calculate direct location of data record on the disk without using index structure. It uses a function, called hash function and generates address when called with search key as parameters. Hash function computes the location of desired data on the disk.

Hash Organization

- **Bucket:** Hash file stores data in bucket format. Bucket is considered a unit of storage. Bucket typically stores one complete disk block, which in turn can store one or more records.
- **Hash Function:** A hash function h, is a mapping function that maps all set of search-keys K to the address where actual records are placed. It is a function from search keys to bucket addresses.

Static Hashing

In static hashing, when a search-key value is provided the hash function always computes the same address. For example, if mod-4 hash function is used then it shall generate only 5 values. The output address shall always be same for that function. The numbers of buckets provided remain same at all times.

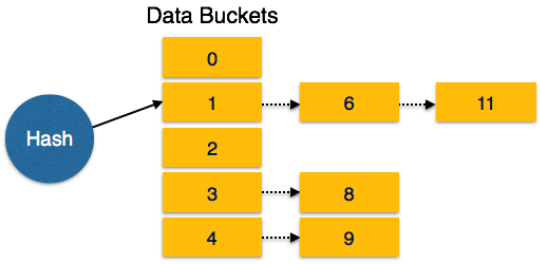


Operation:

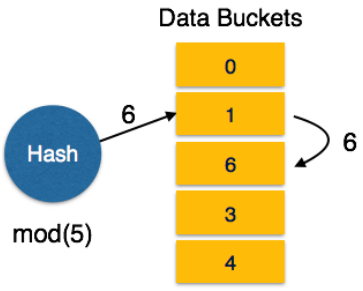
- **Insertion:** When a record is required to be entered using static hash, the hash function h, computes the bucket address for search key K, where the record will be stored.
  Bucket address = h(K)
- **Search:** When a record needs to be retrieved the same hash function can be used to retrieve the address of bucket where the data is stored.
- **Delete:** This is simply search followed by deletion operation.

Bucket Overflow: The condition of bucket-overflow is known as collision. This is a fatal state for any static hash function. In this case overflow chaining can be used.

- **Overflow Chaining:** When buckets are full, a new bucket is allocated for the same hash result and is linked after the previous one. This mechanism is called Closed Hashing.

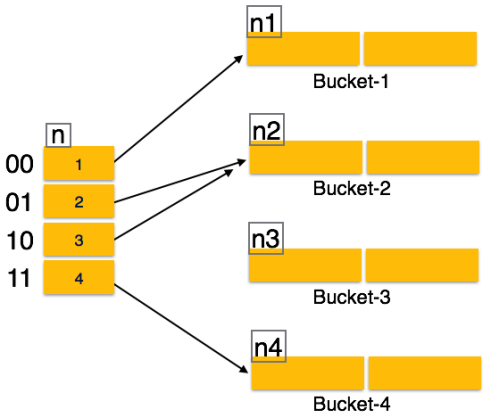


- **Linear Probing:** When hash function generates an address at which data is already stored, the next free bucket is allocated to it. This mechanism is called Open Hashing.
  For a hash function to work efficiently and effectively the following must match:
  - Distribution of records should be uniform
  - Distribution should be random instead of any ordering



Dynamic Hashing

Problem with static hashing is that it does not expand or shrink dynamically as the size of database grows or shrinks. Dynamic hashing provides a mechanism in which data buckets are added and removed dynamically and on-demand. Dynamic hashing is also known as extended hashing. Hash function, in dynamic hashing, is made to produce large number of values and only a few are used initially.



Organization

The prefix of entire hash value is taken as hash index. Only a portion of hash value is used for computing bucket addresses. Every hash index has a depth value, which tells it how many bits are used for computing hash function. These bits are capable to address 2n buckets. When all these bits are consumed, that is, all buckets are full, then the depth value is increased linearly and twice the buckets are allocated.

Operation

- **Querying:** Look at the depth value of hash index and use those bits to compute the bucket address.
- **Update:** Perform a query as above and update data.
- **Deletion:** Perform a query to locate desired data and delete data.
- **Insertion:** compute the address of bucket
  - If the bucket is already full
    - Add more buckets
    - Add additional bit to hash value
    - Re-compute the hash function
  - Else
    - Add data to the bucket
  - If all buckets are full, perform the remedies of static hashing.

Hashing is not favorable when the data is organized in some ordering and queries require range of data. When data is discrete and random, hash performs the best. Hashing algorithm and implementation have high complexity than indexing. All hash operations are done in constant time.