# Module - IV

## Data Storage and Querying

### Q1. Give short note on RAID and different RAID levels.

A Redundant Array of Inexpensive Disks (RAID) may be used to increase disk reliability. The RAID is a collection of variety of disk-organization techniques proposed to achieve improved performance and reliability. The characteristics of these design architectures are :

- RAID is a set of physical disk drives viewed by the operating system as a single logical drive.
- Data are stored across multiple physical disk drives of an array for improved *reliability.*
- Redundant disk capacity is used to store parity information, which guarantees data *recoverability* in case of a disk failure.
- With multiple disks, data are distributed across multiple disks using a scheme known as **data striping** for an improved data transfer rate.

RAID achieve improved data reliability via redundancy. This is achieved by introducing **redundancy**; that is, by storing extra information that can be used in the event of failure of a disk to rebuild the lost information. The simplest (but most expensive) approach to introducing redundancy is to duplicate every disk. This technique is called **mirroring** (or, sometimes, *shadowing*). A logical disk then consists of two physical disks, and every write is carried out on both disks. If one of the disks fails, the data can be read from the other. Data will be lost only if the second disk fails before the first failed disk is repaired.

RAID also benefited by improved performance via parallelism. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk. With multiple disks, we can improve the transfer rate as well (or instead) by striping data across multiple disks. In its simplest form, data striping consists of splitting the bits of each byte across multiple disks; such striping is called **bit-level striping**. That is, if there are an array of eight disks, we write bit $i$ of each byte to disk $i$. **Block-level striping** stripes blocks across multiple disks. It treats the array of disks as a single large disk, and it gives blocks logical numbers. With an array of $n$ disks, block-level striping assigns logical block $i$ of the disk array to disk $(i \bmod n) + 1$; it uses the $\lfloor i/n \rfloor$ th physical block of the disk to store logical block $i$.

#### RAID Levels

The RAID levels are described as follows:

**RAID Level 0:** RAID level 0 refers to disk arrays with striping at the level of blocks, but without any redundancy (such as parity bits). Figure 21(a) shows an array of size 4.

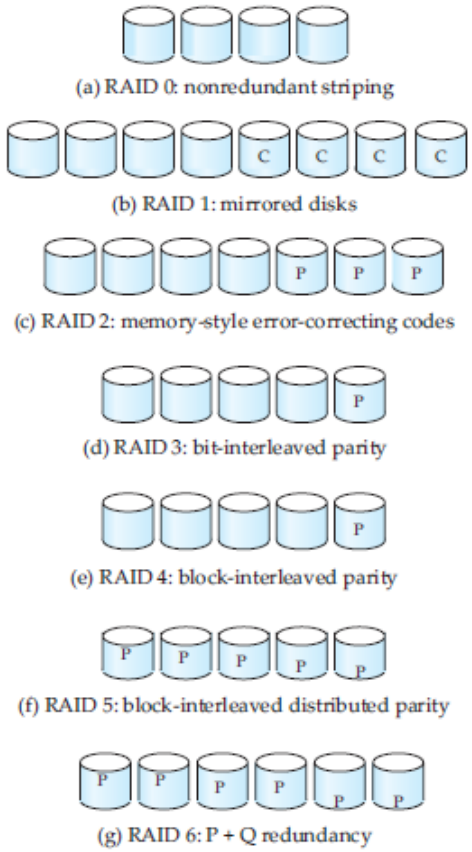



Figure 21: RAID levels. Here $P$ indicates error-correcting bits and $C$ indicates a second copy of the data.

**RAID Level 1:** RAID level 1 refers to disk mirroring with block striping. Figure 21(b) shows a mirrored organization that holds four disks' worth of data.

**RAID Level 2:** RAID level 2 is also known as **memory-style error-correcting code (ECC)** organization. Each byte in a memory system may have a parity bit associated with it that records whether the numbers of bits in the byte set to 1 is even (parity=0) or odd (parity=1). The idea of ECC can be used directly in disk arrays via striping of bytes across disks. If one of the bits in the byte gets damaged (either a 1 becomes a 0, or a 0 becomes a 1), the parity of the byte changes and thus will not match the stored parity. Similarly, if the stored parity bit gets damaged, it will not match the computed parity. Thus, all 1-bit errors will be detected by the memory system.

**RAID level 3:** RAID level 3, or **bit-interleaved parity organization**, improves on level 2 by noting that, disk controllers can detect whether a sector has been read correctly, so a single parity bit can be used for error correction, as well as for detection. The idea is as follows: If one of the sectors gets damaged, we know exactly which sector it is, and, for each bit in the sector, we can figure out whether it is a 1 or a 0 by computing the parity of the corresponding bits from sectors in the other disks. If the parity of the remaining bits is equal to the stored parity, the missing bit is 0; otherwise, it is 1.

**RAID Level 4:** RAID level 4 or **block-interleaved parity organization** uses block-level strip-

ing, as in RAID 0 and in addition keeps a parity block on a separate disk for corresponding blocks from $N$ other disks. This scheme is shown in Figure 21(e). If one of the disks fails, the parity block can be used with the corresponding blocks from the other disks to restore the blocks of the failed disk.

A block read accesses only one disk, allowing other requests to be processed by the other disks. Thus, the data-transfer rate for each access is slower, but multiple read accesses can proceed in parallel, leading to a higher overall I/O rate. Small independent writes also cannot be performed in parallel.

**RAID level 5:** RAID level 5 or **block-interleaved distributed parity** is similar as level 4 but level 5 spreading data and parity among all $N + 1$ disks, rather than storing data in N disks and parity in one disk. For each block, one of the disks stores the parity, and the others store data. By spreading the parity across all the disks in the set, RAID 5 avoids the potential overuse of a single parity disk that can occur with RAID 4.

In level 5, all disks can participate in satisfying read requests, unlike RAID level 4, where the parity disk cannot participate, so level 5 increases the total number of requests that can be met in a given amount of time.

**RAID Level 6:** RAID level 6 (is also called the P+Q redundancy scheme) is much like RAID level 5, but stores extra redundant information to guard against multiple disk failures. Instead of using parity, error-correcting codes such as the **Reed-Solomon codes** are used.

## Q2. Compare fixed-length records and variable-length records.

### Fixed-Length Records

A fixed-length field representation uses the same number of bytes to hold each value of the field in a record. Simple approach is to store each record $i$ starting from byte $n \times (i - 1)$, where $n$ is the size of each record. The advantage of fixed-length records is that the record access is simple but records may cross blocks. That is, part of the record will be stored in one block and the other part in another. It would thus require two block accesses to read or write such a record. To avoid this problem, allocate only as many records to a block as would fit entirely in the block.

For deletion of records, three methods can be employed. One method is to move records $i + 1, ..., n$ to $i, ..., n - 1$. Since this approach requires moving a large number of records another easier method is to simply move the final record of the file into the space occupied by the deleted record. These two approaches are shown in Figure 22(a) and 22(b).

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

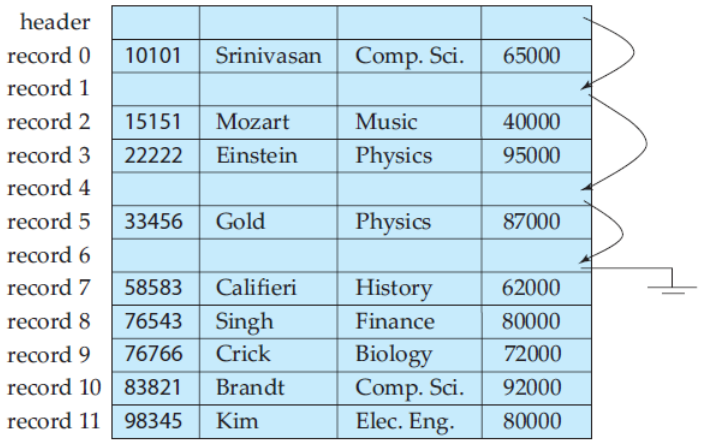
(a) First approach with record 3 deleted and all records moved

| | | | | |
|---|---|---|---|---|
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | 12121 | Wu | Finance | 90000 |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |
| record 4 | 32343 | El Said | History | 60000 |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | 45565 | Katz | Comp. Sci. | 75000 |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |

(b) Second approach with record 3 deleted and final record moved

Figure 22: Two simple approaches that can be employed for deleting fixed-length records

Another approach is to maintain a file header at the beginning of the file and store the address of the first deleted record in the file header. Use this first record to store the address of the second deleted record, and so on. These stored addresses can be used as *pointers*, since they point to the location of a record. The deleted records thus form a linked list, which is often referred to as a **free list** (See Figure 23).



| | | | | |
|---|---|---|---|---|
| header | | | | |
| record 0 | 10101 | Srinivasan | Comp. Sci. | 65000 |
| record 1 | | | | |
| record 2 | 15151 | Mozart | Music | 40000 |
| record 3 | 22222 | Einstein | Physics | 95000 |
| record 4 | | | | |
| record 5 | 33456 | Gold | Physics | 87000 |
| record 6 | | | | |
| record 7 | 58583 | Califieri | History | 62000 |
| record 8 | 76543 | Singh | Finance | 80000 |
| record 9 | 76766 | Crick | Biology | 72000 |
| record 10 | 83821 | Brandt | Comp. Sci. | 92000 |
| record 11 | 98345 | Kim | Elec. Eng. | 80000 |

Figure 23: Third approach with free list, after deletion of records 1, 4, and 6.

### Variable-Length Records

A variable-length field representation uses different sizes for storing records in a file. In contrast to fixed-length records this representation is memory efficient. 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.
- Record types that allow repeating fields, such as arrays or multisets.

The representation of a record with variable-length attributes typically has two parts: an initial part with fixed length attributes, followed by data for variable length attributes. Fixed-length attributes, such as numeric values, dates, or fixed length character strings are allocated as many bytes as required to store their value. Variable-length attributes, such as varchar types, are represented in the initial part of the record by a pair (*offset*, *length*), where *offset* denotes where the data for that attribute begins within the record, and *length* is the length in bytes of the variable-sized attribute. The values for these attributes are stored consecutively, after the initial fixed-length part of the record. Thus, the initial part of the record stores a fixed size of information about each attribute, whether it is fixed-length or variable-length. An example of such a record representation is shown in Figure 24.

For storing variable-length records in a block, the **slotted-page structure** is used. The organization of records within a block is shown in Figure 25. There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header.
2. The end of free space in the block.

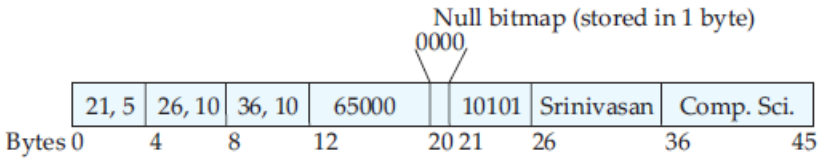



Figure 24: Representation of variable-length record.

3. 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 a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.

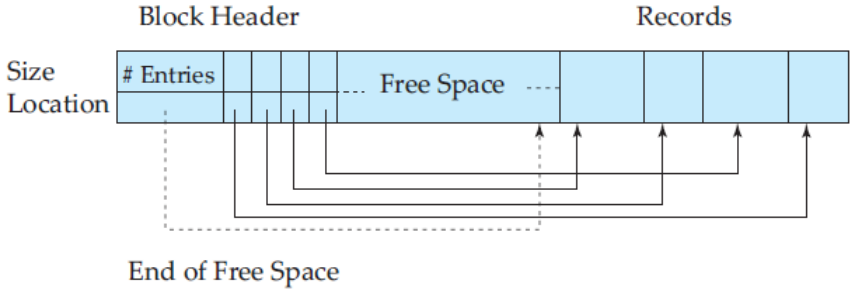



Figure 25: Slotted-page structure.

If a record is deleted, the space that it occupies is freed, and its entry is set to *deleted* and the records are moved to occupy the freed space. End-of-free-space pointer in the header is appropriately updated as well.

The slotted-page structure requires that there will be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record.

## Q3. Explain sequential file organization.

A sequential file is designed for efficient processing of records in sorted order based on some search key. 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. Figure 26 shows an example sequential file organization.

The sequential file organization allows records to be read in sorted order; that can be useful for certain query-processing. However, it is difficult to maintain physical sequential order as records are inserted and deleted frequently. For insertion following rules can be applied:

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.

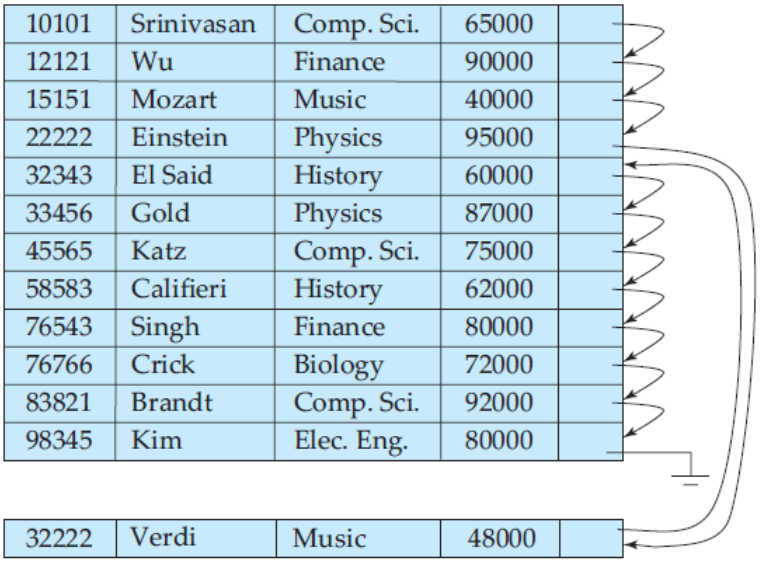



Figure 26: Sequential file organization.

Deletion of records can be managed by using pointer chains.

If relatively few records need to be stored in overflow blocks, this approach works well. However, the correspondence between search-key order and physical order may be totally lost over a period of time, in which case sequential processing will become much less efficient. At this point, the file should be *reorganized* so that it is once again physically in sequential order and these reorganizations are costly.

## Q4. Explain multitable clustering file organization.

Many relational database systems store each relation in a separate file, so that they can take full advantage of the file system that the operating system provides. However, many large-scale database systems do not rely directly on the underlying operating system for file management. Instead, one large operating system file is allocated to the database system. The database system stores all relations in this one file, and manages the file itself.

A **multitable clustering file organization** is a file organization, such as that illustrated in Figure 27, that stores related records of two or more relations in each block. Such a file organization allows us to read records that would satisfy the join condition by using one block read. Thus, we are able to process this particular query more efficiently.

| Comp. Sci. | Taylor | 100000 |
|---|---|---|
| 45564 | Katz | 75000 |
| 10101 | Srinivasan | 65000 |
| 83821 | Brandt | 92000 |
| Physics | Watson | 70000 |
| 33456 | Gold | 87000 |

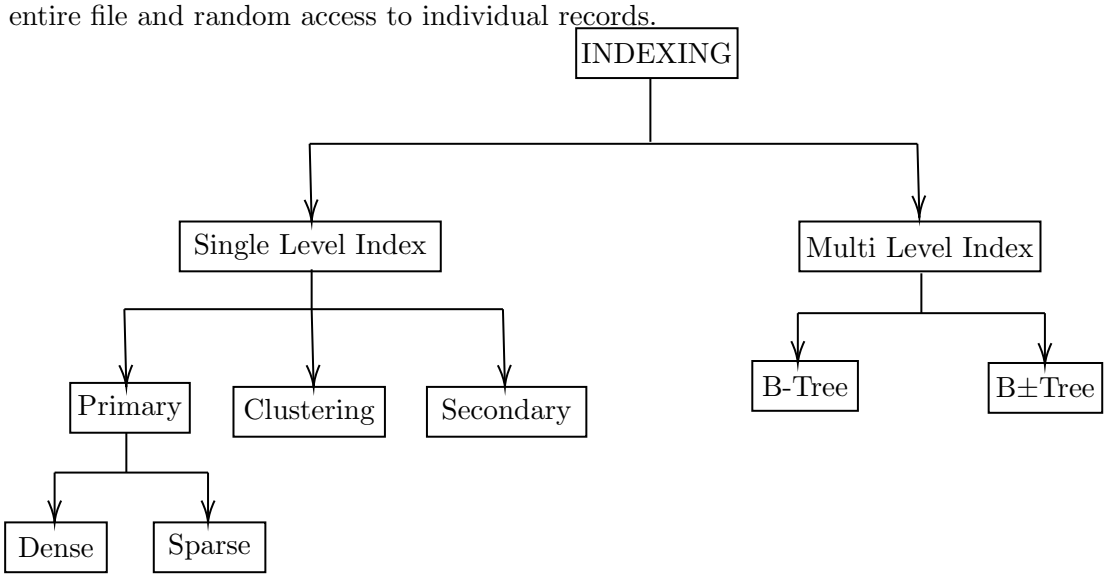
Figure 27: Multitable clustering file structure.

# Indexing and Hashing

## Q5. What is indexing? Explain different types of indexing.

Indexes are data structures that are used to speed up access to desired data. The index structures are additional files on disk that provide secondary access paths, which provide alternate ways to access the records without affecting the physical placement of records in the data file on disk. An index file consists of records (called **index entries**) of the form (search key, pointer) where **search key** is an attribute or a set of attributes used to look up records in a file and **pointer** is the address to the disk block that contain records with the search key value. Index files are typically much smaller than the original data file and the values in the index are **ordered** so that a binary search can be performed on index for faster search.

A data file that are ordered sequentially on some search key, with a clustering 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.




### Primary Index

A **primary index** is an ordered file whose records are of fixed length with two fields, and it acts like an access structure to efficiently search for and access data records in a data file. The first field is of the same data type as the ordering key field - called the **primary key** - of the data file, and the second field is a pointer to a disk block. There is one index entry in the index file for each block in the data file. Each index entry has the value of primary key field for the first record in a block and a pointer to that block as its two field values. The total number of entries in the index is the same as the number of disk blocks in the ordered data file.

### Clustering Indexes

If file records are physically ordered on a non-key field — which does not have a distinct value for each record — that field is called the **clustering field** and the data file is called a **clustered file**. In this case, a different type of index, called a **clustering index**, ca be created to speed up retrieval of all the records that have the same value for the clustering field. This differs from a

primary index, which requires that the ordering field of the data file have a distinct value for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a disk block pointer. There is one entry in the clustering index for each distinct value of the clustering field, and it contains the value and a pointer to the first block in the data file that has a record with that value for its clustering field. An example clustered index file is shown in Figure 28.

Record insertion and deletion cause problems because the data records are physically ordered. To alleviate the problem of insertion, it is common to reserve a whole block (or a cluster of contiguous blocks) for each value of the clustering field; all records with that value are placed in the block (or block cluster). This makes insertion and deletion relatively straightforward.

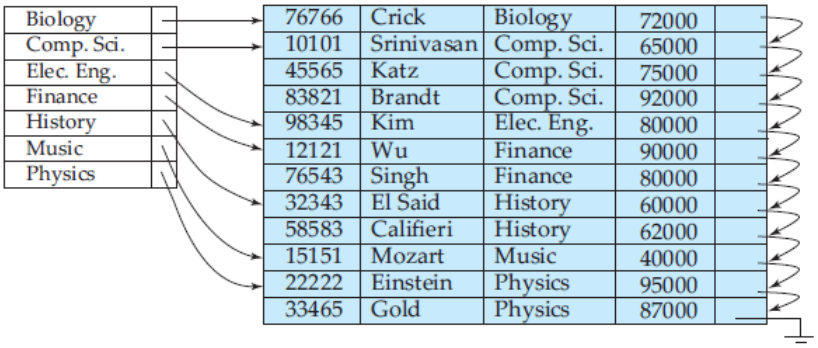



Figure 28: A example of clustered dense index file.

### Secondary Indexes

A secondary index provides a secondary means of accessing a data file for which some primary access already exists. The data file records could be ordered, unordered, or hashed. The secondary index may be created on a field that is a candidate key and has a unique value in every record, or on a non-key field with duplicate values. The index is again an ordered file with two fields. The first field is of the same data type as some nonordering field of the data file that is an indexing field. The second field is either a block pointer or a record pointer.

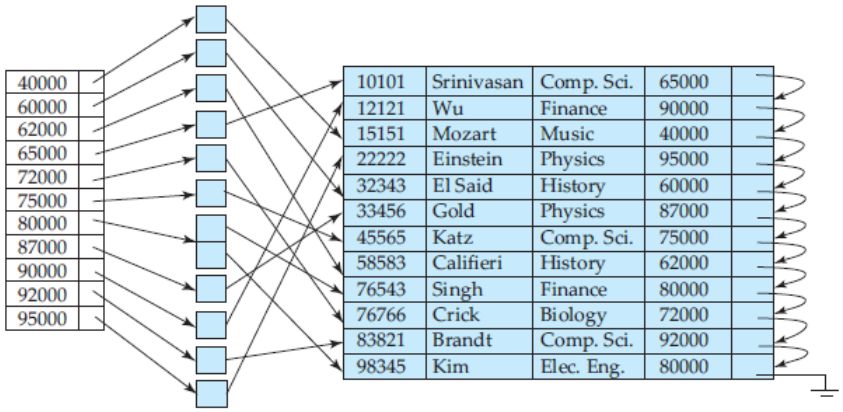



Figure 29: Secondary index on non-candidate key field.

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 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. One such representation is shown in Figure 29.

It needs 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. Secondary indices improve the performance of queries that use keys other than the search key of the clustering index. However, they impose a significant overhead on modification of the database.

### Multi-level indexes

If an index is small enough to be kept entirely in main memory, the search time to find an entry is low. However, if the index is so large that not all of it can be kept in memory, index blocks must be fetched from disk when required. The search for an entry in the index then requires several disk-block reads. In such a case, it is always desirable to create yet another level of index. Indeed, this process can be repeated as many times as necessary. Indices with two or more levels are called **multilevel indices**.

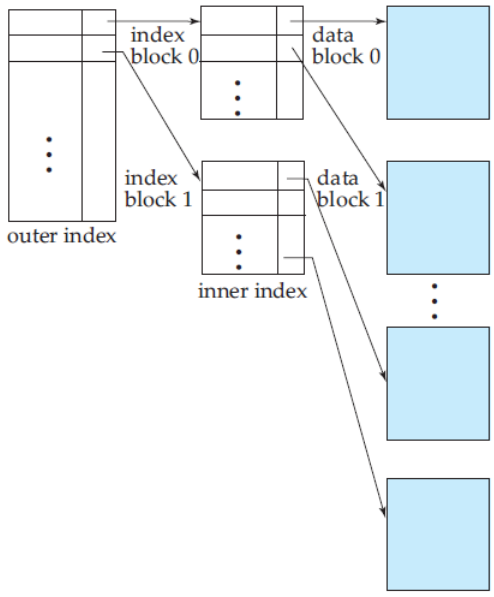



Figure 30: Two-level sparse index.

To create multi-level indexes, treat the original index just as any other sequential file, and construct a sparse *outer index* on the original index, called as the *inner index*, as shown in Figure 30. Note that the index entries are always in sorted order, allowing the outer index to be sparse. To locate a record, first use a binary search on the outer index to find the record for the largest search-key value less than or equal to the one that need to be located. The pointer points to a block of the inner index. Scan this block until the record is found that has the largest search-key value less than or equal to the desired record. The pointer in this record points to the block of the

file that contains the actual record.

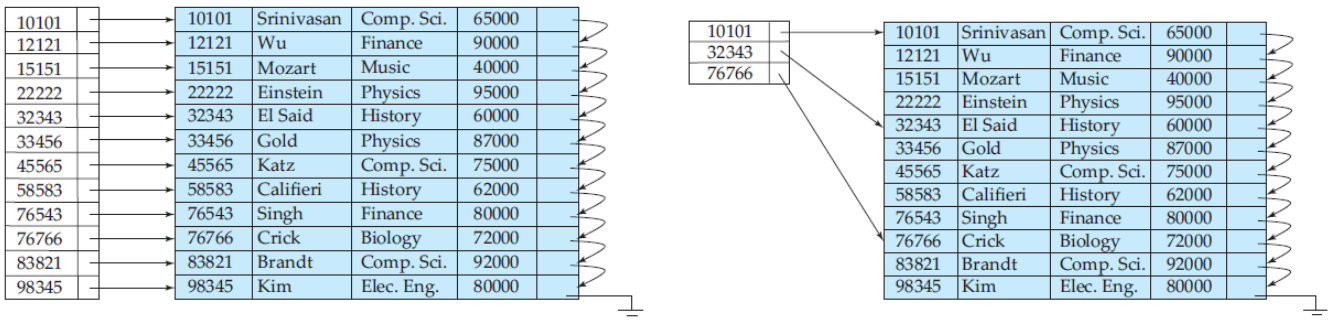
Searching for records with a multilevel index requires significantly fewer I/O operations than does searching for records by binary search. Multilevel indices are closely related to tree structures, such as the binary trees used for in-memory indexing.

## Q6. Explain dense and sparse indices.

An **index entry**, or **index record**, consists of a search-key value and pointers to one or more records with that value as their search-key value. The pointer to a record consists of the identifier of a disk block and an offset within the disk block to identify the record within the block.

**Dense index:** In a dense index, an index entry appears for every search-key value in the file. In a dense clustering 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 clustering one, records are sorted on the same search key. In a dense nonclustering index, the index must store a list of pointers to all records with the same search-key value.

**Sparse index:** In a sparse index, an index entry appears for only some of the search-key values. Sparse indices can be used only if the relation is stored in sorted order of the search key. Each index entry contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, an index entry with the largest search-key value that is less than or equal to the search-key value must be determined first. Then, start at the record pointed to by that index entry, and follow the pointers in the file until the desired record is found.




(a) (b)

Figure 31: (a) Dense index. (b) Sparse index.

## Q7. Explain B+ Tree and its structure.

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 nonleaf node in the tree has between $\lceil n/2 \rceil$ and $n$ children, where $n$ is fixed for a particular tree.

### Structure of a 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. Figure 32 shows a typical node of a $B^+$-tree. It contains up to $n-1$ search-key values $K_1, K_2, ..., K_{n-1}$, and $n$ pointers $P_1, P_2, ..., P_n$. The search-key values within a node are kept in sorted order; thus, if $i < j$, then $K_i < K_j$ .

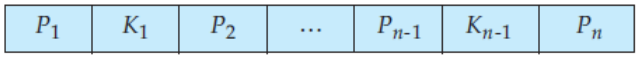



Figure 32: Typical node of a B+-tree.

In a $B^+$-tree, data pointers are stored only at the **leaf nodes** of the tree; hence, the structure of leaf nodes differs from the structure of internal nodes. In a leaf node, a pointer $P_i$, for $i = 1, 2, ..., n-1$, points to a file record with search-key value $K_i$. Since there is a linear order on the leaves based on the search-key values that they contain, the pointer $P_n$ is used to chain together the leaf nodes in search-key order. This ordering allows for efficient sequential processing of the file. This is shown in Figure 34.

Each leaf node can hold up to $n-1$ values, though in practice it is better to contain as few as $\lceil (n-1)/2 \rceil$ values. The ranges of values in each leaf do not overlap, except if there are duplicate search-key values, in which case a value may be present in more than one leaf. Specifically, if $L_i$ and $L_j$ are leaf nodes and $i < j$, then every search-key value in $L_i$ is less than or equal to every search-key value in $L_j$. If the $B^+$-tree index is used as a dense index every search-key value must appear in some leaf node. A typical leaf node is shown in Figure 33.

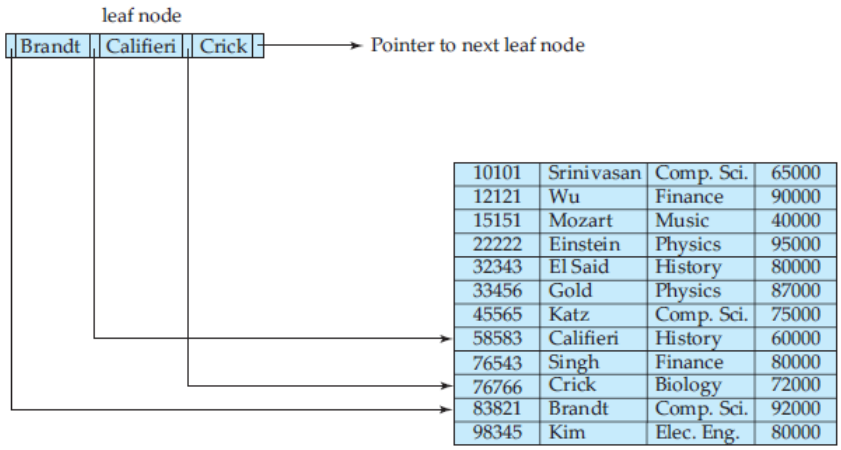



Figure 33: A leaf node in $B^+$-tree index ($n = 4$).

The **nonleaf nodes** of the $B^+$-tree form a multilevel (sparse) index on the leaf nodes. The structure of nonleaf nodes is the same as that for leaf nodes, except that all pointers are pointers to tree nodes. A nonleaf node may hold up to n pointers, and must hold at least $\lceil n/2 \rceil$ pointers. Nonleaf nodes are also referred to as **internal nodes**. Unlike other nonleaf nodes, the root node can hold fewer than $\lceil n/2 \rceil$ pointers; however, it must hold at least two pointers, unless the tree consists of only one node.

All $B^+$-trees are balanced. That is, the length of every path from the root to a leaf node is the same. This property is a requirement for a $B^+$- tree. Indeed, the "B" in $B^+$-tree stands

for "balanced." It is the balance property of B$^+$-trees that ensures good performance for lookup, insertion, and deletion.

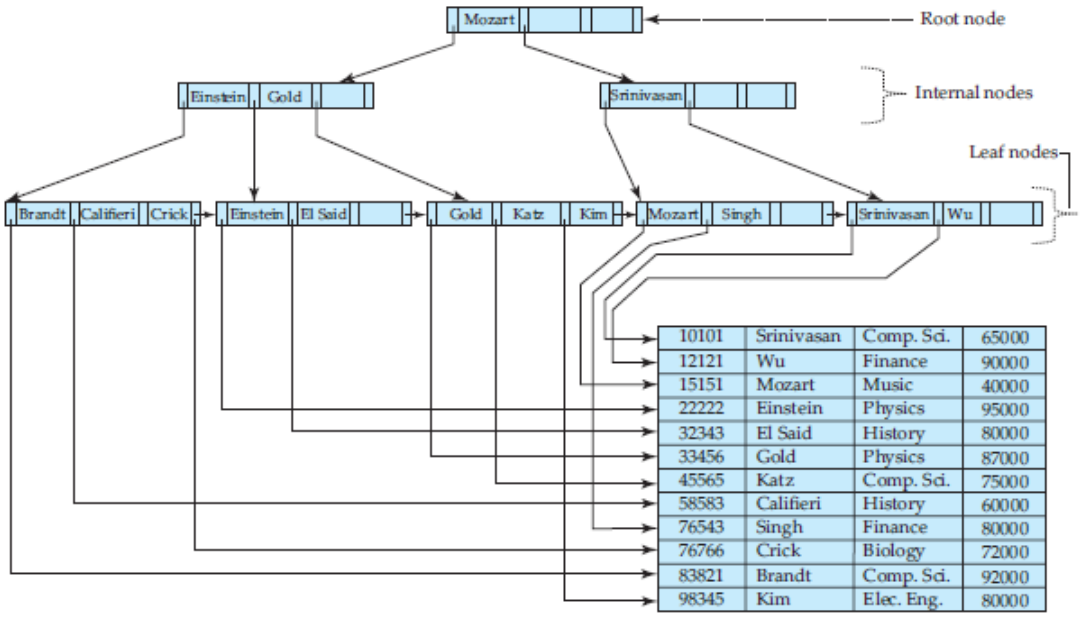



Figure 34: A example B$^+$-tree index ($n = 4$).

## Q8. Explain B Tree and its structure.

**B-tree** indices are similar to B$^+$-tree indices. The primary distinction between the two approaches is that a B-tree eliminates the redundant storage of search-key values. That is, in B+-tree every search-key value appears in some leaf node; several are repeated in nonleaf nodes.

A B-tree allows search-key values to appear only once (if they are unique), unlike a B$^+$-tree, where a value may appear in a nonleaf node, in addition to appearing in a leaf node. Figure 35 shows a B-tree representation. Since search keys are not repeated in the B-tree, it may be able to store the index in fewer tree nodes than in the corresponding B$^+$-tree index. However, since search keys that appear in nonleaf nodes appear nowhere else in the B-tree, we are forced to include an additional pointer field for each search key in a nonleaf node. These additional pointers point to either file records or buckets for the associated search key.

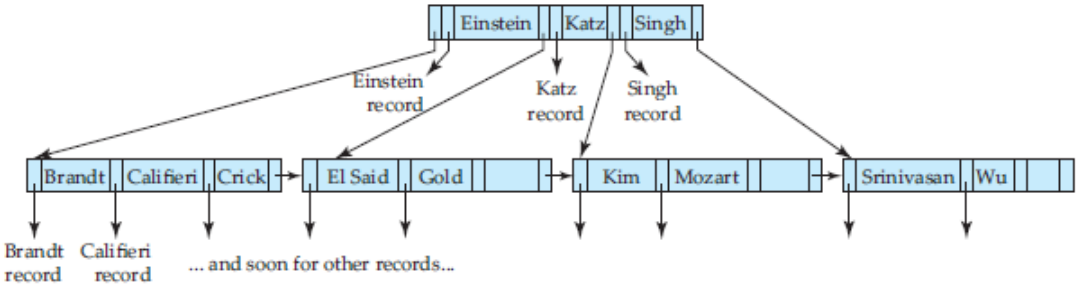



Figure 35: A example B-tree.

The number of nodes accessed in a lookup in a B-tree depends on where the search key is

located. A lookup on a $B^+$-tree requires traversal of a path from the root of the tree to some leaf node. In contrast, it is sometimes possible to find the desired value in a B-tree before reaching a leaf node.

## Q9. Explain hashing. Discuss on static hashing and dynamic hashing.

One disadvantage of sequential file organization is that we must access an index structure to locate data, or must use binary search, and that results in more I/O operations. File organizations based on the technique of **hashing** allow us to avoid accessing an index structure. Hashing also provides a way of constructing indices. Hashing use **bucket** as a unit of storage that can store one or more records. A bucket is typically a disk block, but could be chosen to be smaller or larger than a disk block.

Let $K$ denote the set of all search-key values, and let $B$ denote the set of all bucket addresses. A hash function $h$ in hashing is a function from $K$ to $B$. Let $h$ denote a hash function. To insert a record with search key $K_i$, the hash function $h(K_i)$ will be computed, which gives the address of the bucket for that record and the record is stored in that bucket. To perform a lookup on a search-key value $K_i$ , again compute $h(K_i)$, then search the bucket with that address. Deletion is equally straightforward. If the search-key value of the record to be deleted is $K_i$, compute $h(K_i)$, then search the corresponding bucket for that record, and delete the record from the bucket.

It is obvious that a hash function that maps many different key values to a single address or one that does not map the key values uniformly is a bad hash function. Typical hash functions perform computation on the internal binary machine representation of characters in the search key. A simple hash function of this type first computes the sum of the binary representations of the characters of a key, then returns the sum modulo the number of buckets. Figure 36 shows a hash organization, with eight buckets.

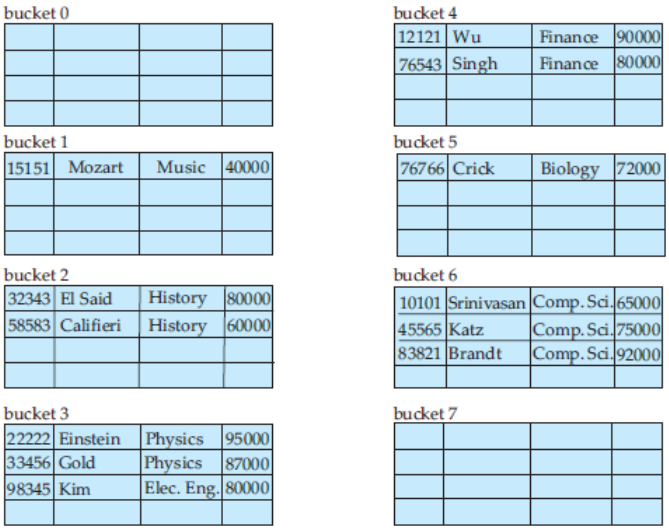



Figure 36: An example hash organization.

### Static Hashing

In **static hashing**, when a search-key value is provided the hash function always computes the same address. For example, if $i$ 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.

The condition of **bucket-overflow** is known as **collision**. This is a fatal state for any static hash function. Bucket overflow can occur because of:

- Insufficient buckets
- Skew: Skew in distribution of records can occur due to two reasons:
  - multiple records have same search-key value
  - chosen hash function produces non-uniform distribution of key values

In static hashing bucket overflow is handled using a linked list called **overflow chaining**. In 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**. In an 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 bucket. One policy is to use the next bucket (in cyclic order) that has space; this policy is called *linear probing*. Other policies, such as computing further hash functions, are also used.

An important drawback of static hashing is that the hash function chosen while implementing the system is fixed, and it cannot be changed. Since the function $h$ maps search-key values to a fixed set $B$ of bucket addresses, wastage of space will occur if $B$ is made large. At the same time, if $B$ is too small, bucket overflows can occur. As the file grows, performance suffers.

### Dynamic Hashing

Static hashing technique will fail for databases that grows and shrinks in size over time. So, **dynamic hashing** techniques allow the hash function to be modified dynamically to accommodate the growth or shrinkage of the database. One such kind of technique is known as **extensible hashing**.

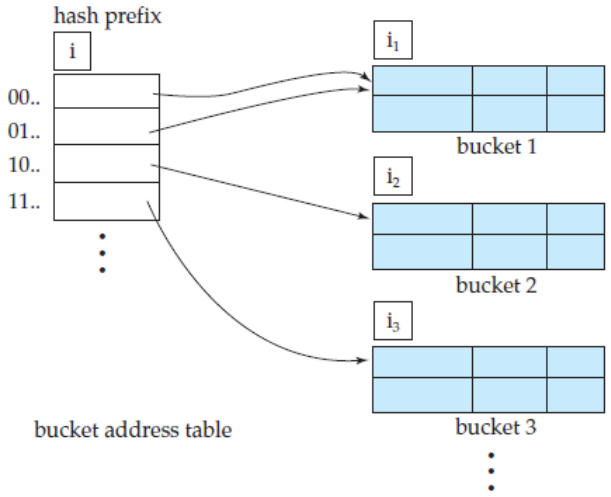



Figure 37: General extendable hash structure.

With extendable hashing, the hash function generates values over a relatively large range—namely, $b$-bit binary integers. A typical value for $b$ is 32. This scheme do not create a bucket for each hash

value. Instead, it use only i bits of the hash function to index into a table of bucket addresses where $0 \leq i \leq b$. These $i$ bits are used as an offset into an additional table of bucket addresses. The value of $i$ grows and shrinks with the size of the database. Figure 37 shows a general extendable hash structure.

The $i$ appearing above the bucket address table in the Figure 37 indicates that $i$ bits of the hash value $h(K)$ are required to determine the correct bucket for $K$. This number change as the file grows. Multiple entries in the bucket address table may point to a bucket. Each bucket $j$ stores a value $i_j$; all the entries that point to the same bucket have the same values on the first $i_j$ bit. The number of bucket-address-table entries that point to bucket $j$ is $2^{(i-i_j)}$.