**MODULE IV**

**RAID**

RAID, or “**Redundant Arrays of Independent Disks**” is a technique which makes use of a combination of multiple disks instead of using a single disk for increased performance, data redundancy or both. The term was coined by David Patterson, Garth A. Gibson, and Randy Katz at the University of California, Berkeley in 1987

It consists of an array of disks in which multiple disks are connected to achieve different goals.

**Why data redundancy?**

Data redundancy, although taking up extra space, adds to disk reliability. This means, in case of disk failure, if the same data is also backed up onto another disk, we can retrieve the data and go on with the operation. On the other hand, if the data is spread across just multiple disks without the RAID technique, the loss of a single disk can affect the entire data.

**Key evaluation points for a RAID System**

* **Reliability:**How many disk faults can the system tolerate?
* **Availability:** What fraction of the total session time is a system in uptime mode, i.e. how available is the system for actual use?
* **Performance:**How good is the response time? How high is the throughput (rate of processing work)? Note that performance contains a lot of parameters and not just the two.
* **Capacity:** Given a set of N disks each with B blocks, how much useful capacity is available to the user?

RAID is very transparent to the underlying system. This means, to the host system, it appears as a single big disk presenting itself as a linear array of blocks. This allows older technologies to be replaced by RAID without making too many changes in the existing code.

**RAID technology**

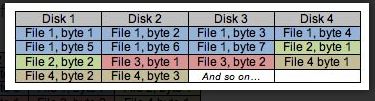
There are 7 levels of RAID schemes. These schemas are as RAID 0, RAID 1, ...., RAID 6.

These levels contain the following characteristics:

* It contains a set of physical disk drives.
* In this technology, the operating system views these separate disks as a single logical disk.
* In this technology, data is distributed across the physical drives of the array.
* Redundancy disk capacity is used to store parity information.
* In case of disk failure, the parity information can be helped to recover the data.

### Striping

### Most RAID levels employ striping, it is a term used to describe when individual files are split and written to more than one disk. Striping is the way that RAID gets around the performance limitation of mechanical storage by performing read and write operations to all disks simultaneously.



Striping can be done at **byte level or block level**. **Byte-level striping** means that each file is split up into parts **one byte in size**. Using the same 4 disk array as an example, the first byte would be written to the first drive, the second byte to the second drive and so on, until the fifth byte is then written to the first drive again and the whole process starts over.

In the case of **block level striping**, the above table can simply be repeated with the word ‘**block’ substituted for ‘byte’**. Each file is split into parts one block in size, which is 512 bytes by default, but can be specified otherwise. The size of this block is commonly referred to as stripe size. If a file is smaller than the stripe size, it simply gets stored on a single disk.

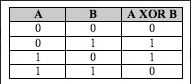
Striping alone does not include redundancy, and therefore includes no protection against data loss. Many RAID levels use mirroring or parity alongside striping to ensure data security and availability.

**Mirroring/Shadowing**

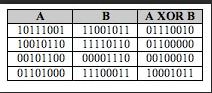
Mirroring is the simplest way to give **redundant storage.** This is a technique that can only be used on two disks unless combined with striping. When data is written to one disk, it is simultaneously written to the other disk, so in a mirrored array the two drives are always an exact copy of each other. If one of the drives fails, service can continue uninterrupted and without data loss as the other drive simply takes over. This method of redundancy doesn’t require any fancy calculations so is usually a part of onboard RAID solutions as it’s quite cheap to implement. The down side of mirroring is the inefficient use of space. In an array that uses mirroring, half of the total capacity of the disks goes to redundancy.

**Parity**

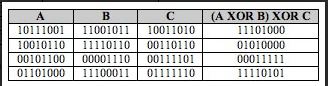
Parity can be used alongside striping as a way to offer redundancy without losing half of the total capacity to duplicate data. In an array using parity for redundancy, as little as one disk’s worth of space is needed to store the information needed to recover the entire array in the case of a disk failure. This parity information is derived from the data stored on the disk by using the logical ‘exclusive OR’ operation.



The exclusive OR operation, or XOR operation, when applied to two true/false values is true if, and only if, only one of the values is true. Since we’re talking computers here, instead of true and false we’re using ones and zeroes - zero for false, one for true.



In the above table, bytes are used as opposed to single bits. Even though random values were entered into the first two columns, if one of them was erased, the information can be recreated using the XOR operation on the two remaining columns.



**Standard RAID levels**

**RAID 0**

* RAID level 0 provides data stripping, i.e., a data can place across multiple disks. It is based on stripping that means if one disk fails then all data in the array is lost.
* This level doesn't provide fault tolerance but increases the system performance.

Example:

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** |
| 20 | 21 | 22 | 23 |
| 24 | 25 | 26 | 27 |
| 28 | 29 | 30 | 31 |
| 32 | 33 | 34 | 35 |

In this figure, block 0, 1, 2, 3 form a stripe.

In this level, instead of placing just one block into a disk at a time, we can work with two or more blocks placed it into a disk before moving on to the next one.

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** |
| 20 | 22 | 24 | 26 |
| 21 | 23 | 25 | 27 |
| 28 | 30 | 32 | 34 |
| 29 | 31 | 33 | 35 |

In this above figure, there is no duplication of data. Hence, a block once lost cannot be recovered.

**Pros of RAID 0:**

* In this level, throughput is increased because multiple data requests probably not on the same disk.
* This level full utilizes the disk space and provides high performance.
* It requires minimum 2 drives.

**Cons of RAID 0:**

* It doesn't contain any error detection mechanism.
* The RAID 0 is not a true RAID because it is not fault-tolerance.
* In this level, failure of either disk results in complete data loss in respective array.

**RAID 1**

This level is called mirroring of data as it copies the data from drive 1 to drive 2. It provides 100% redundancy in case of a failure.

**Example:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** |
| A | A | B | B |
| C | C | D | D |
| E | E | F | F |
| G | G | H | H |

Only half space of the drive is used to store the data. The other half of drive is just a mirror to the already stored data.

**Pros of RAID 1:**

The main advantage of RAID 1 is fault tolerance. In this level, if one disk fails, then the other automatically takes over.

* In this level, the array will function even if any one of the drives fails.

**Cons of RAID 1:**

* In this level, one extra drive is required per drive for mirroring, so the expense is higher.

**RAID 2**

* RAID 2 consists of bit-level striping using hamming code parity. In this level, each data bit in a word is recorded on a separate disk and ECC code of data words is stored on different set disks.
* Due to its high cost and complex structure, this level is not commercially used. This same performance can be achieved by RAID 3 at a lower cost.

**Pros of RAID 2:**

* This level uses one designated drive to store parity.
* It uses the hamming code for error detection.

**Cons of RAID 2:**

* It requires an additional drive for error detection.

**RAID 3**

* RAID 3 consists of byte-level striping with dedicated parity. In this level, the parity information is stored for each disk section and written to a dedicated parity drive.
* In case of drive failure, the parity drive is accessed, and data is reconstructed from the remaining devices. Once the failed drive is replaced, the missing data can be restored on the new drive.
* In this level, data can be transferred in bulk. Thus high-speed data transmission is possible.

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** |
| A | B | C | P(A, B, C) |
| D | E | F | P(D, E, F) |
| G | H | I | P(G, H, I) |
| J | K | L | P(J, K, L) |

**Pros of RAID 3:**

* In this level, data is regenerated using parity drive.
* It contains high data transfer rates.
* In this level, data is accessed in parallel.

**Cons of RAID 3:**

* It required an additional drive for parity.
* It gives a slow performance for operating on small sized files.

**RAID 4**

* RAID 4 consists of block-level stripping with a parity disk. Instead of duplicating data, the RAID 4 adopts a parity-based approach.
* This level allows recovery of at most 1 disk failure due to the way parity works. In this level, if more than one disk fails, then there is no way to recover the data.
* Level 3 and level 4 both are required at least three disks to implement RAID.

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** |
| A | B | C | P0 |
| D | E | F | P1 |
| G | H | I | P2 |
| J | K | L | P3 |

In this figure, we can observe one disk dedicated to parity.

In this level, parity can be calculated using an XOR function. If the data bits are 0,0,0,1 then the parity bits is XOR(0,1,0,0) = 1. If the parity bits are 0,0,1,1 then the parity bit is XOR(0,0,1,1)= 0. That means, even number of one results in parity 0 and an odd number of one results in parity 1.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **C1** | **C2** | **C3** | **C4** | **Parity** |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |

Suppose that in the above figure, C2 is lost due to some disk failure. Then using the values of all the other columns and the parity bit, we can recompute the data bit stored in C2. This level allows us to recover lost data.

**RAID 5**

* RAID 5 is a slight modification of the RAID 4 system. The only difference is that in RAID 5, the parity rotates among the drives.
* It consists of block-level striping with DISTRIBUTED parity.
* Same as RAID 4, this level allows recovery of at most 1 disk failure. If more than one disk fails, then there is no way for data recovery.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Disk 0** | **Disk 1** | **Disk 2** | **Disk 3** | **Disk 4** |
| 0 | 1 | 2 | 3 | P0 |
| 5 | 6 | 7 | P1 | 4 |
| 10 | 11 | P2 | 8 | 9 |
| 15 | P3 | 12 | 13 | 14 |
| P4 | 16 | 17 | 18 | 19 |

This figure shows that how parity bit rotates.

This level was introduced to make the random write performance better.

**Pros of RAID 5**:

* This level is cost effective and provides high performance.
* In this level, parity is distributed across the disks in an array.
* It is used to make the random write performance better.

**Cons of RAID 5:**

* In this level, disk failure recovery takes longer time as parity has to be calculated from all available drives.
* This level cannot survive in concurrent drive failure.

**RAID 6**

* This level is an extension of RAID 5. It contains block-level stripping with 2 parity bits.
* In RAID 6, you can survive 2 concurrent disk failures. Suppose you are using RAID 5, and RAID 1. When your disks fail, you need to replace the failed disk because if simultaneously another disk fails then you won't be able to recover any of the data, so in this case RAID 6 plays its part where you can survive two concurrent disk failures before you run out of options.

|  |  |  |  |
| --- | --- | --- | --- |
| **Disk 1** | **Disk 2** | **Disk 3** | **Disk 4** |
| A0 | B0 | Q0 | P0 |
| A1 | Q1 | P1 | D1 |
| Q2 | P2 | C2 | D2 |
| P3 | B3 | C3 | Q3 |

**Pros of RAID 6:**

* This level performs RAID 0 to strip data and RAID 1 to mirror. In this level, stripping is performed before mirroring.
* In this level, drives required should be multiple of 2.

**Cons of RAID 6:**

* It is not utilized 100% disk capability as half is used for mirroring.
* It contains very limited scalability.

**FILE ORGANIZATION**

A file organization is a way of arranging the records in a file when the file is stored on secondary storage (disk, tape etc.). The different ways of arranging the records enable different operations to be carried out efficiently over the file. A database management system supports several file organization techniques. The most important task of DBA is to choose a best organization for each file, based on its use. The organization of records in a file is influenced by number of factors that must be taken into consideration while choosing a particular technique. These factors are (a) fast retrival, updation and transfer of records, (b) efficient use of disk space, (c) high throughput, (d) type of use, (e) efficient manipulation, (f) security from unauthorized access, (g) scalability, (h) reduction in cost, (i) protection from failure.

**Basic Concepts of Files**

A file is organized logically as a **sequence of records**. Each file is also logically partitioned into fixed-length storage units called **blocks**, which are the units of both storage allocation and data transfer. Most databases use block sizes of 4 to 8 kilobytes by default, but many databases allow the block size to be specified when a database instance is created. Larger block sizes can be useful in some database applications.

**Records and Record Types**

Data is generally stored in the form of **records.** A record is a collection of fields or data items and data items is formed of one or more bytes. Each record has a unique identifier called record-id. The records in a file are one of the following two types:

1. Fixed length records.
2. Variable length records.

In addition, we shall require that each record is entirely contained in a single block; that is, no record is contained partly in one block, and partly in another. This restriction simplifies and speeds up access to data items.

**Fixed Length Records**

Every record in the file has exactly the same size (in bytes). The record slots are uniform and are arranged in a continuous manner in the file. A record is identified using both record-id and slot number f the record.

type instructor = record

ID varchar (5);

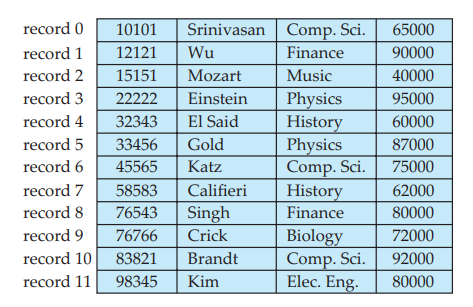
name varchar(20);

dept name varchar (20);

salary numeric (8,2);

end

Assume that each character occupies 1 byte and that numeric (8,2) occupies 8 bytes. Suppose that instead of allocating a variable amount of bytes for the attributes ID, name, and dept name, we allocate the maximum number of bytes that each attribute can hold. Then, the instructor record is 53 bytes long. A simple approach is to use the first 53 bytes for the first record, the next 53 bytes for the second record, and so on (FIGURE 1)

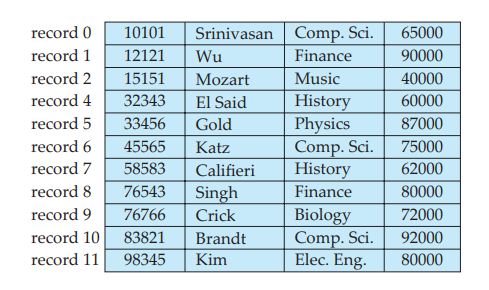


**Figure 1: File containing instructor records**

1. Unless the block size happens to be a multiple of 53 (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.
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.

To avoid the first problem, we allocate only as many records to a block as would fit entirely in the block (this number can be computed easily by dividing the block size by the record size, and discarding the fractional part). Any remaining bytes of each block are left unused.

When a 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 (Figure 2). Such an approach requires moving a large number of records. It might be easier simply to move the final record of the file into the space occupied by the deleted record (Figure 3).

****

**Figure 2:** File of Figure 1, with Record 3 deleted and all records moved.

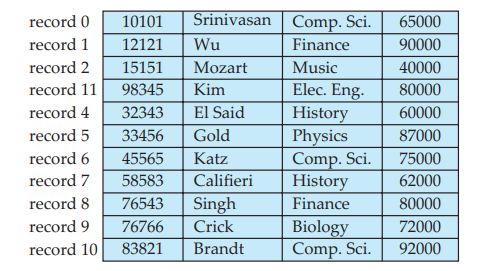
****

Figure 3: File of Figure 2, with record 3 deleted and final record moved.

It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses. Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space. A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done. Thus, we need to introduce an additional structure.

At the beginning of the file, we allocate a certain number of bytes as a file header. The header will contain a variety of information about the file. For now, all we need to store there is the address of the first record whose contents are deleted. We use this first record to store the address of the second available record, and so on. Intuitively, we can think of these stored addresses 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. Figure 4 shows the file of Figure1, with the free list, after records 1, 4, and 6 have been deleted.

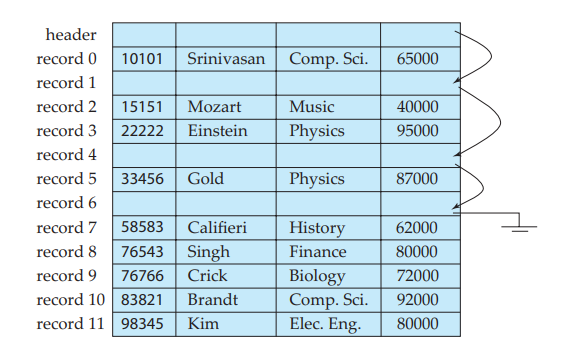


Figure 4: File of Figure 1, with free list after deletion of records 1, 4, and 6.

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 record to the end of the file. Insertion and deletion for files of fixed-length records are simple to implement, because the space made available by a deleted record is exactly the space needed to insert a record. If we allow records of variable length in a file, this match no longer holds. An inserted record may not fit in the space left free by a deleted record, or it may fill only part of that space.

**Variable-Length Records**

Every record in the file need **not be of the same size** (in bytes). Therefore, the records in the file have different sizes. The major problem with variable length record is that when a new record is to be inserted, an empty slot of the exact length is required. **If the slot is smaller, it cannot be used and if it is too big, the extra space is just wasted.**

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.

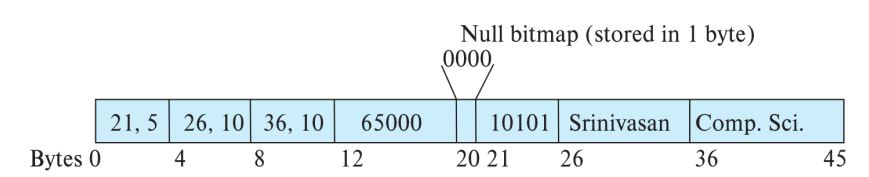
• Record types that allow repeating fields, such as arrays or multisets.

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.

**Record representation**

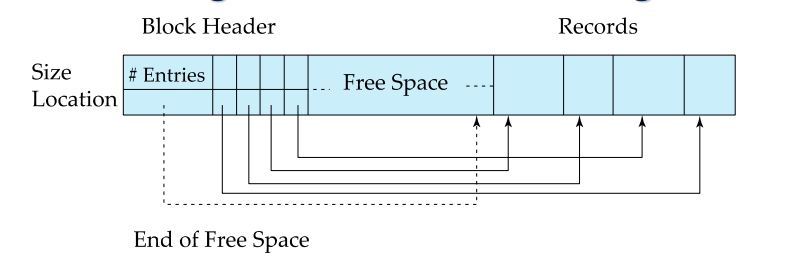
****

* Attributes are stored in order
* Variable length attributes represented by fixed size (offset, length), with actual data stored after all fixed length attributes
* Null values represented by null-value bitmap.

The figure shows an instructor record, whose first three attributes ID, name, and dept name are variable-length strings, and whose fourth attribute salary is a fixed-sized number. We assume that the offset and length values are stored in two bytes each, for a total of 4 bytes per attribute. The salary attribute is assumed to be stored in 8 bytes, and each string takes as many bytes as it has characters.

The figure also illustrates the use of a null **bitmap**, which indicates which attributes of the record have a null value. In this particular record, if the salary were null, the fourth bit of the bitmap would be set to 1, and the salary value stored in bytes 12 through 19 would be ignored. Since the record has four attributes, the null bitmap for this record fits in 1 byte, although more bytes may be required with more attributes.

The **slotted-page structure is commonly used for organizing records within a block,** and is shown in Figure.

****

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.

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.

Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.

Pointers should not point directly to record — instead they should point to the entry for the record in header.

Databases often store data that can be much larger than a disk block. For instance, an image or an audio recording may be multiple megabytes in size, while a video object may be multiple gigabytes in size. Recall that SQL supports the types **blob and clob,** which store binary and character large objects.

Most relational databases restrict the size of a record to be no larger than the size of a block, to simplify buffer management and free-space management.

**FILE ORGANIZATION TECHNIQUES**

A file organization is a way of arranging the records in a file when the file is stored on secondary storage (disk, tape etc). There are different types of file organizations that are used by applications. The operations to be performed and the selection of storage device are the major factors that influence the choice of a particular file organization. The different types of file organizations are as follows:

**Heap file organization**. Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation.

**Sequential – store** records in sequential order, based on the value of the **search key** of each record.

**Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed

Records of each relation may be stored in a separate file. In a multitable **clustering file organization** records of several different relations can be stored in the same file.

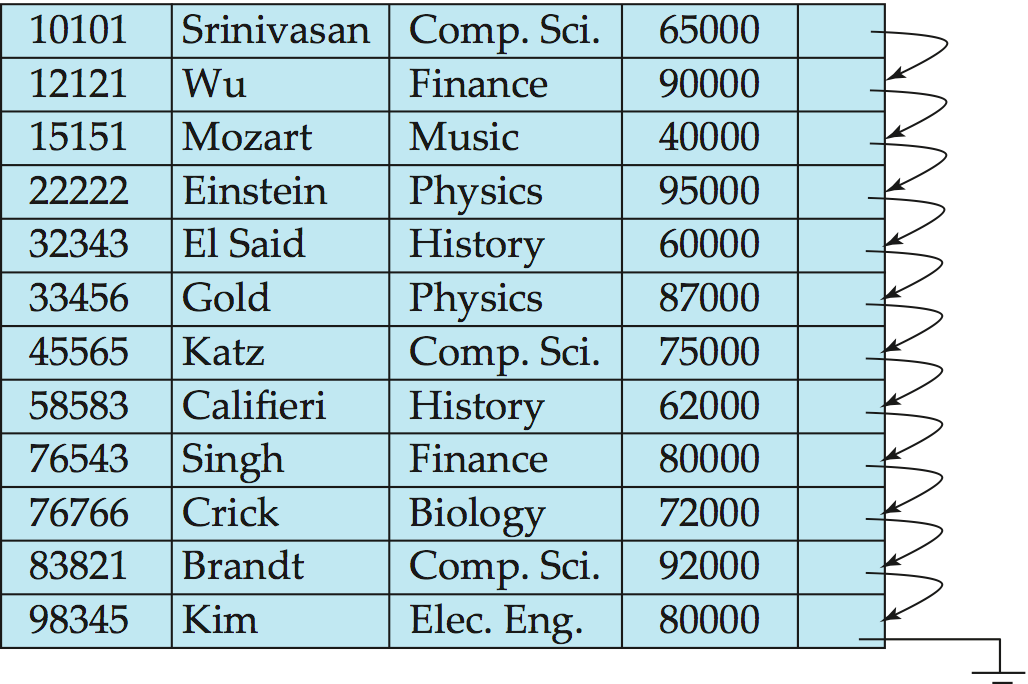
Motivation: store related records on the same block to minimize I/O

**Sequential File Organization**

In sequential file organization, records are stored in a sequential order according to the “search key”. A **Search key** is an attribute or a set of attributes which are used **to serialize the records**. It is not necessary that search key must be primary key.

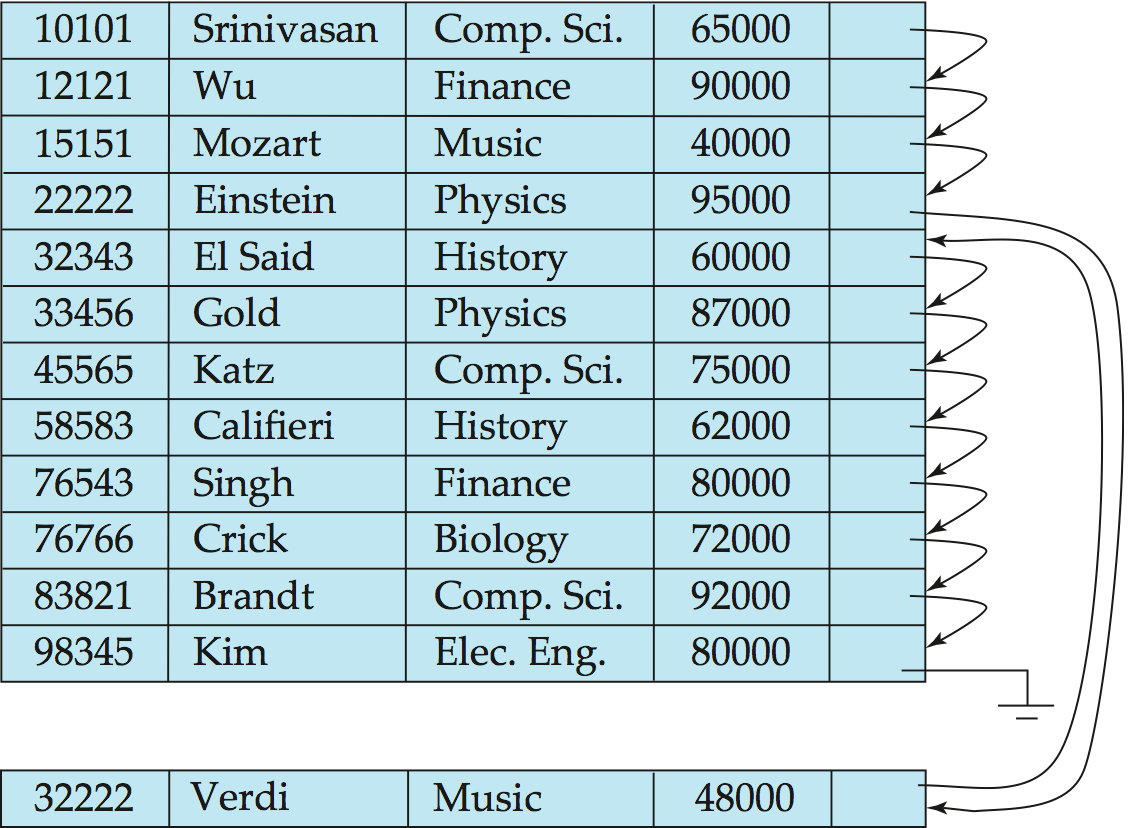
To permit fast retrieval of records in search-key order, we chain together records by **pointers.** The pointer in each record points to the next record in search-key order. Furthermore, to minimize the number of block accesses in sequential file processing, we store records physically in search-key order, or as close to search-key order as possible.

Figure 1 shows a sequential file of instructor records taken from our university example. In that example, the records are stored in search-key order, using ID as the search key



**Figure 1: Sequential file for instructor records.**

* 1. Locate the record in the file that comes before the record to be inserted in search-key order.
  2. 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

****

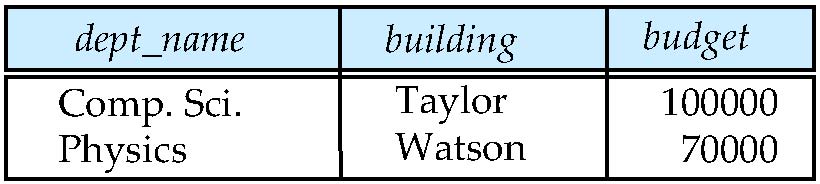
**Figure 2:** Sequential file after an insertion

* **Deletion –** use pointer chains
* **Insertion –**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

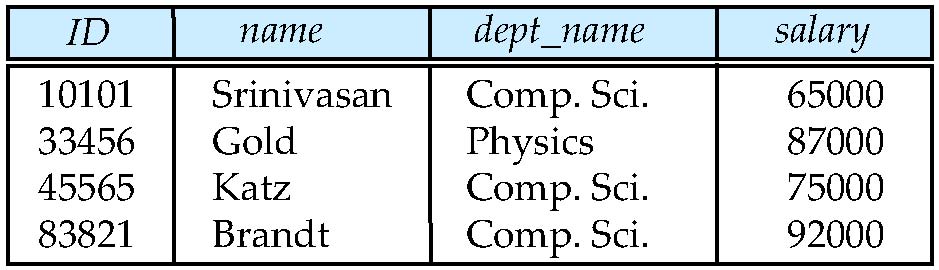
**Multitable Clustering File Organization**

**Store several relations in one file using a multitable clustering file organization**

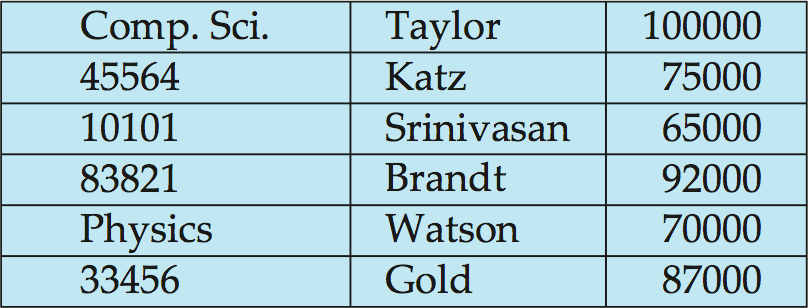
***Department***

****

***instructor***

****

**Multitable clustering of *department* andinstructor**

****

**INDEXING**

Indexes are used to speed up the retrieval of records if certain requirements on search conditions are met. An index for a file of records works just like an index catalogue in a library. In a normal library environment, for example, there should be catalogues such as author indexes and book title indexes. A user may use one of these indexes to quickly find the location of a required book, if he/she knows the author(s) or title of the book.

Data is stored in the form of records and every record has a key field, which helps it to be recognizing uniquely. Indexing is a data structure technique to efficiently retrieve records from the database on some attributes on which the indexing has been done. Indexing in database is similar what we see in books.

**Indexing in DBMS**

* Indexing is used to optimize the performance of a database by minimizing the number of disk accesses required when a query is processed.
* The index is a type of data structure. It is used to locate and access the data in a database table quickly.
* It is defined based on the indexing attribute.

The merits of indexes can be measured in the following aspects:

* **Access types**: The kind of access methods that can be supported efficiently (e.g. value-based search or range search).
* **Access time:** Time needed to locate a particular record or a set of record
* **Insertion time**: The time it takes to insert a new data item. This value includes the time it takes to find the correct place to insert the new data item, as well as the time it takes to update the index structure.
* **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.
* **Space overhead**: The additional space occupied by an index structure. Provided that the amount of additional space is moderate, it is usually worthwhile to sacrifice the space to achieve improved performance.

An attribute or set of attributes used to look up records in a file is called a **search key.**

An **index file** consists of records (called **index entries**) of the form

C:\Users\MCET\Pictures\za42.JPG

* The first column of the database is the search key that contains a copy of the primary key or candidate key of the table. The values of the primary key are stored in sorted order so that the corresponding data can be accessed easily.
* The second column of the database is the data reference. It contains a set of pointers holding the address of the disk block where the value of the particular key can be found.

**Ordered Indices**

In ordered indexing, records of file are stored in some sorted order in physical memory. The values in the index are ordered (sorted) so that binary search can be performed on the index.

In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.

Example: Suppose we have an employee table with thousands of record and each of which is 10 bytes long. If their IDs start with 1, 2, 3....and so on and we have to search student with ID-543.

* In the case of a database with no index, we have to search the disk block from starting till it reaches 543. The DBMS will read the record after reading 543\*10=5430 bytes.
* In the case of an index, we will search using indexes and the DBMS will read the record after reading 542\*2= 1084 bytes which are very less compared to the previous case.

Indexing is defined based on its indexing attributes. Indexing can be of the following types –

* **Primary index:** in a sequentially ordered file, the index whose search keys the sequential specifies order of the file. The key field is generally the primary key of the relation.
* Also called **clustering index** to avoid confusion with Primary Key.
* The search key of a primary index is usually, but not necessarily, the primary key.
* **Secondary index**:an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index**.**

## Clustering Index

* A clustered index can be defined as an ordered data file. Sometimes the index is created on non-primary key columns which may not be unique for each record.
* In this case, to identify the record faster, we will group two or more columns to get the unique value and create index out of them. This method is called a clustering index.
* The records which have similar characteristics are grouped, and indexes are created for these group.
* **Index-sequential file:** ordered sequential file with a primary index.

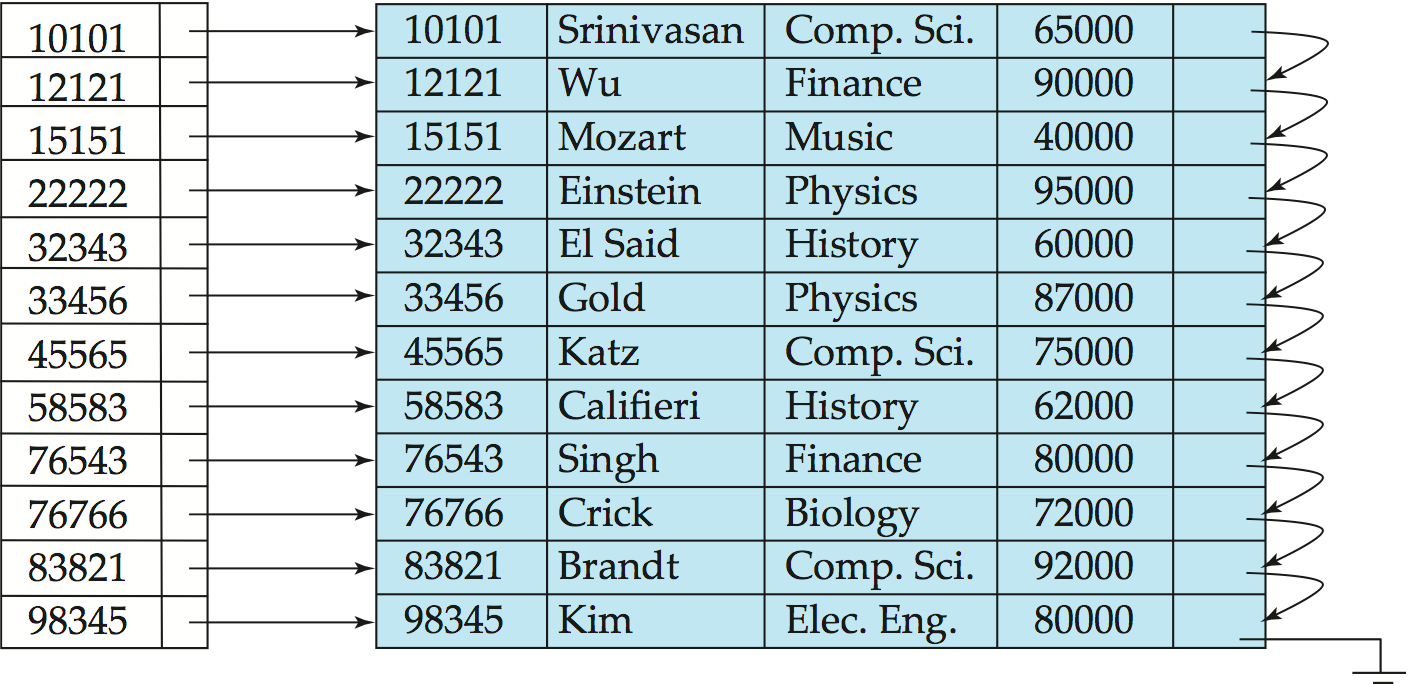
**Dense and Sparse Indices**

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.

There are two types of ordered indices:

**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 non clustering index, the index must store a list of pointers to all records with the same search-key value.

****

**Figure: Dense index**

**Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name***

As another example, suppose that the search-key value is not not a primary key. Figure shows a dense clustering index for the instructor file with the search key being dept name. Observe that in this case the instructor file is sorted on the search key dept name, instead of ID, otherwise the index on dept name would be a nonclustering index. Suppose that we are looking up records for the History department. Using the dense index of Figure, we follow the pointer directly to the first History record. We process this record, and follow the pointer in that record to locate the next record in search-key (dept name) order. We continue processing records until we encounter a record for a department other than History

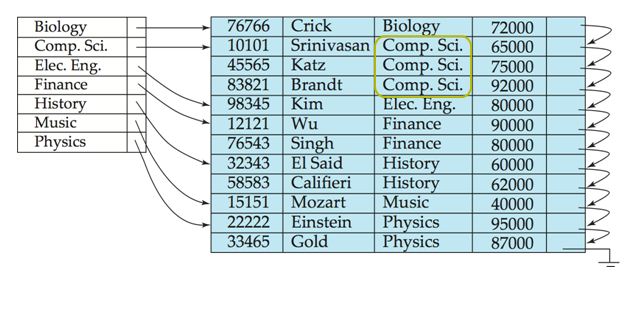
****

Figure : Dense index with search key dept name.

**Advantages of Dense index:**

The advantages of dense index are:

* 1. It is efficient technique for small and medium sized data files.
  2. Searching is comparatively fast and efficient.

**Disadvantages of Dense index:**

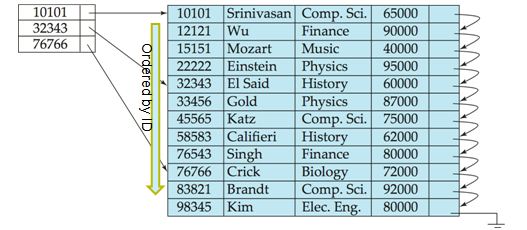
The disadvantages of dense index are:

1. Index table is large and require more memory space.
2. Insertion and deletion is comparatively complex.
3. In-efficient for large data files.

**Sparse index:**

On contrary, in sparse indexing there are only some records in index table for unique values of the search-key attribute of file and a pointer to the first data record with that value. To search a record in sparse index we search for a value that is less than or equal to value in index for which we are looking. After getting the first record, linear search is performed to retrieve the desired record. There is at most one sparse index since it is not possible to build a sparse index that is not clustered.

Suppose that we are looking up the record of instructor with ID “22222”. If we are using the sparse index (**Figure: Sparse index),**we do not find an index entry for “22222”. Since the last entry (in numerical order) before “22222” is “10101”, we follow that pointer. We then read the instructor file in sequential order until we find the desired record.

****

**Figure: Sparse index.**

**Advantages of Sparse index**: The advantages of sparse index are:

* 1. Index table is small and hence save memory space (specially in large files).
  2. Insertion and deletion is comparatively easy.

**Disadvantages of Sparse index**: The disadvantages of sparse index are:

1. Searching is comparatively slower, since index table is searched and then linear search is performed inside secondary memory.

**Single and Multilevel Indexes**

• **Single level indexes:**

A single stage index for a data file is known as single level index. A single level index cannot be divided. It is useful in small and medium size data files. If the file size is bigger, then single level, indexing is not an efficient method. Searching is faster than other indexes for small size data files

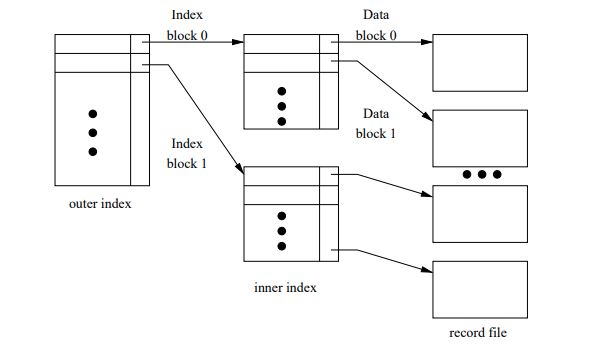
**• Multilevel index:**

* If primary index does not fit in memory, access to records becomes expensive.
* To reduce number of disk accesses to index entries, treat primary index on disk as sequential file and construct a sparse index on it.

– Outer index → a sparse index of primary index

– Inner index → the primary index file

**Multilevel Index structure**

****

* If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
* Note that indexes at all levels must be updated on insertions or deletions of records from a file.

A single index for a large size data file increases the size of index table and increases the search time that results in slower searches. The idea behind multilevel indexes is that, a single level index is divided into multiple levels, which reduces search time.

In multilevel indexes, the first level index consists of two fields, the first field consists of a value of search key attributes and a second field consists of a pointer to the block (or second level index) which consists that value and so on.

To search a record in multilevel index, binary search is used to find the largest of all the small value or equal to the one that needs to be searched. The pointer points to a block of the inner index. After reaching to the desired block, the desired record is searched (in case of two-level indexing) otherwise again the largest of the small values or equal to the one that needs to be searched and so on.

Benefits of multilevel indexes are they reduce search time significantly for large size data files.

**Index Update**

**Single-level indices**

**Insertion.** First, the system performs a lookup using the search-key value that appears in the record to be inserted. The actions the system takes next depend on whether the index is dense or sparse.

**Dense indices:**

1. If the search-key value does not appear in the index, the system inserts an index entry with the search-key value in the index at the appropriate position.
2. Otherwise the following actions are taken:
3. If the index entry stores pointers to all records with the same searchkey value, the system adds a pointer to the new record in the index entry
4. Otherwise, the index entry stores a pointer to only the first record with the search-key value. The system then places the record being inserted after the other records with the same search-key values.

**Sparse indices:**

We assume that the index stores an entry for each block. If the system creates a new block, it inserts the first search-key value (in search-key order) appearing in the new block into the index. On the other hand, if the new record has the least search-key value in its block, the system updates the index entry pointing to the block; if not, the system makes no change to the index.

**Deletion.** To delete a record, the system first looks up the record to be deleted. The actions the system takes next depend on whether the index is dense or sparse:

**Dense indices:**

1. If the deleted record was the only record with its particular search-key value, then the system deletes the corresponding index entry from the index.
2. Otherwise the following actions are taken:
3. If the index entry stores pointers to all records with the same searchkey value, the system deletes the pointer to the deleted record from the index entry.
4. Otherwise, the index entry stores a pointer to only the first record with the search-key value. In this case, if the deleted record was the first record with the search-key value, the system updates the index entry to point to the next record.

**Sparse indices:**

1. If the index does not contain an index entry with the search-key value of the deleted record, nothing needs to be done to the index.
2. Otherwise the system takes the following actions:
   * + 1. If the deleted record was the only record with its search key, the system replaces the corresponding index record with an index record for the next search-key value (in search-key order). If the next search-key value already has an index entry, the entry is deleted instead of being replaced.
       2. Otherwise, if the index entry for the search-key value points to the record being deleted, the system updates the index entry to point to the next record with the same search-key value.