UNIT – VIII

8.1 Data on External Storage

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

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

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

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

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

8.2 File Organization

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

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

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

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

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

8.2.1 Indexing

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

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

The 3 Alternatives for Data Entries in an Index:

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

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

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

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

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

8.2.2 Types of Indexes

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

Ordered Index:

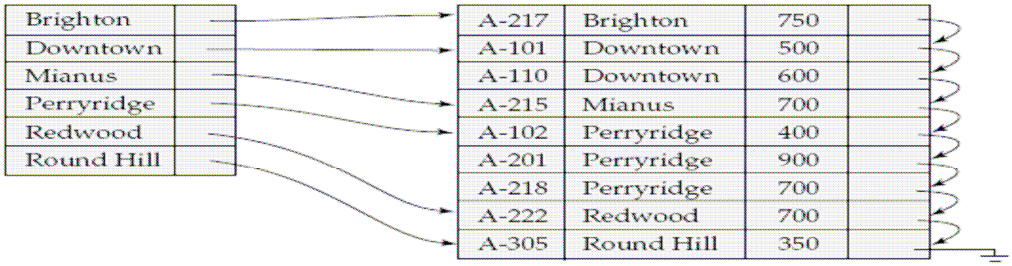
Based on a stored ordering of the values. To gain fast random access to records in a file, we can use an index structure. Each index structure is associated with a particular search key. Just like the index of a book or a library catalog, an ordered index stores the values of the search keys in sorted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

Primary Index:

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

Dense index:

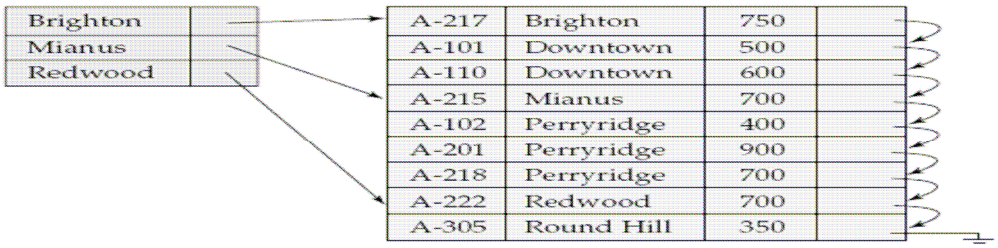
An index record appears for every search-key value in the file. In a dense primary index, the index record contains the search-key value and a pointer to the first data record with that search-key value. The rest of the records with the same search key-value would be stored sequentially after the first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



Sparse index:

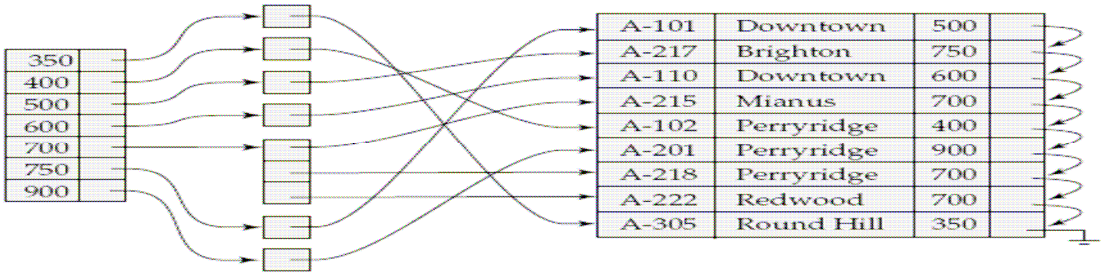
An index record appears for only some of the search-key values. As is true in dense indices, each index record contains a search-key value and a pointer to the first data record with that search-key value. To locate a record, we find the index entry with the largest search-key value that is less

than or equal to the search-key value for which we are looking. We start at the record pointed to by that index entry, and follow the pointers in the file until we find the desired record.



Secondary Index:

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



Hash Index:

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

h(ki) = h(k4)

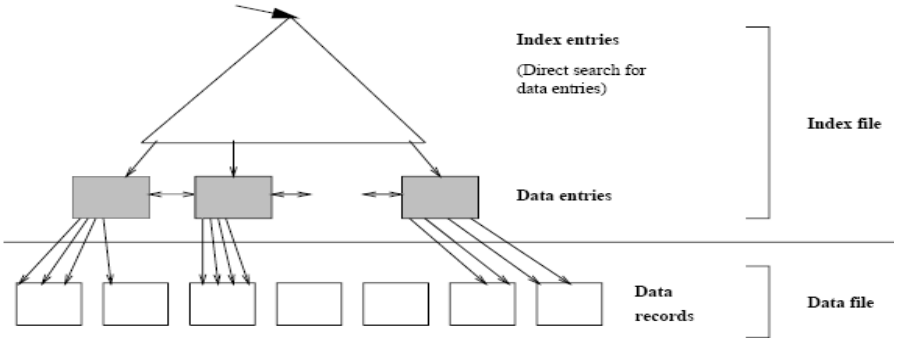
Where h = Hash function.

B = Set of bucket addresses

K = Set of search key.

Clustered Index & Unclustered Index:

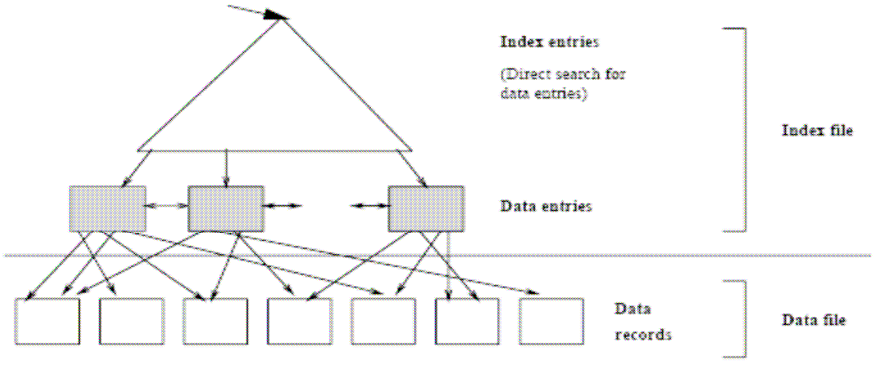
When a file is organized so that the ordering of data records is the same as or close to the ordering of data entries in some index, we say that the index is clustered. Other wise it is Unclustered. An index that uses Alternative (1) is clustered, by definition. An index that uses Alternative (2) or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



Clustered Index

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

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



Unclustered index

Index Data Structures

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

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

1. Hash Based Indexing:

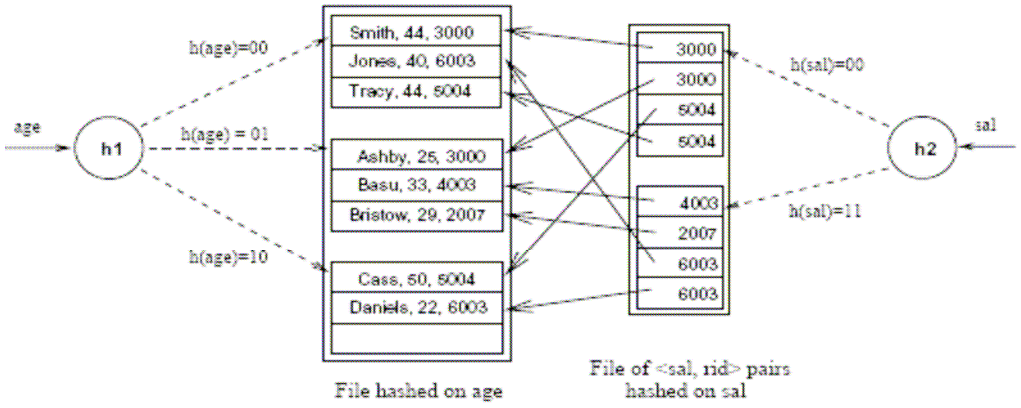
This type of indexing is used to find records quickly, by providing a search key value. We can organize records using a technique called hashing to quickly find records that have a given search key value. In this approach the records are grouped in Buckets. Here the Bucket contains the Primary page and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

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

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

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

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



File hashed on Age, with Index on Sal

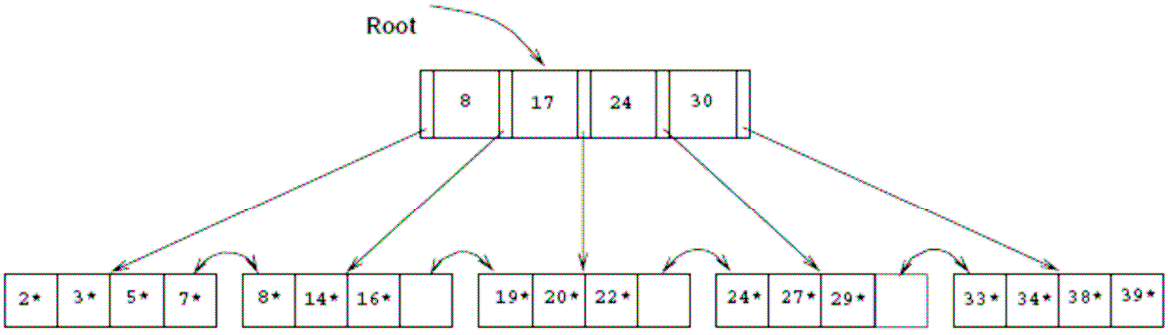
2. Tree – Based Indexing:

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

Example:

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

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



Tree – Structured Index

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

Comparison of File Organizations

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

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

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

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

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

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

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

Cost Model:

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

Notations used in this model:

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

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

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

C = The average time needed for a Record processes.

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

F = Fan – out (in Tree Indexes).

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

Observation to be Note:

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

1. Heap Files:

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

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

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

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

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

$$B(D+RC)$$

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

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

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

2. Sorted Files:

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

Cost of Scanning Sorted files = Cost of scanning Heap Files

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

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

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

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

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

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

Which is same as the cost of insertion, and includes

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

3. Clustered Files:

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

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

Search with equality selection:

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

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

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

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

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

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

I/O Costs Comparison

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

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

Choosing a File Organization:

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

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

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

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

Indexes and Performance Tuning

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

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

1. Equality Selection
2. Range Selection

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

Advantages of using Tree structured indexes:

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

Disadvantages:

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

Clustered index organization:

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

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

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

Index only-evaluation:

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

Advantages: It uses only Unclustered indexes.

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

Clustered Indexes-Example:

Consider the following the database query:

SELECT s.rno from student s

WHERE s.marks > 70.

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

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

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

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

Use of Aggregate Operators:

Consider the given example

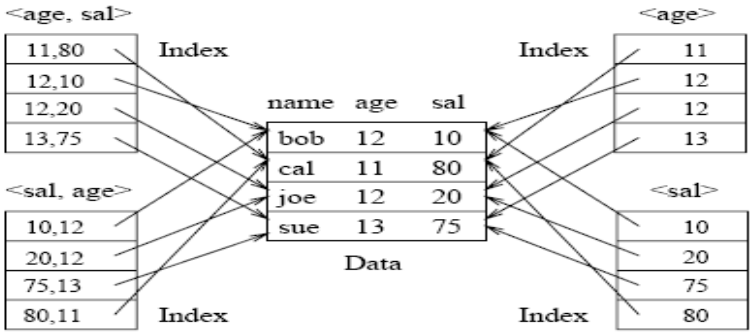
SELECT s.sno, COUNT(*) from student s GROUP BY s.sno.

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

Composite Search Keys:

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

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



Composite Search key Index

Index Specification In SQL:

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

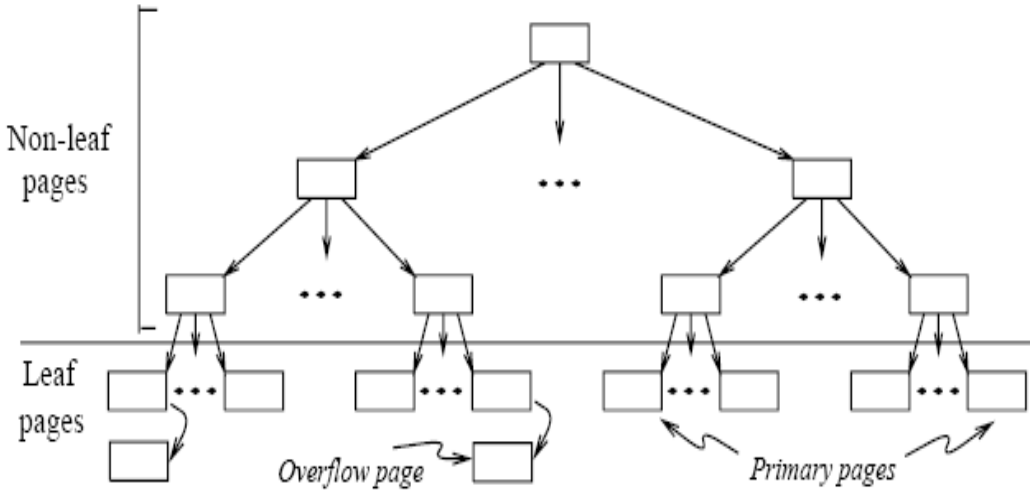
```
CREATE INDEX IndAgeRating ON Students
WITH STRUCTURE = BTREE,
KEY = (age, gpa)
```

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

8.7 Indexed Sequential Access Methods (ISAM)

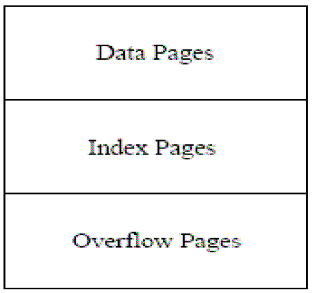
The ISAM is static method. To understand the motivation for the ISAM technique, it is useful to begin with a simple sorted file.

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



Structure of ISAM

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



Page Allocation in ISAM

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

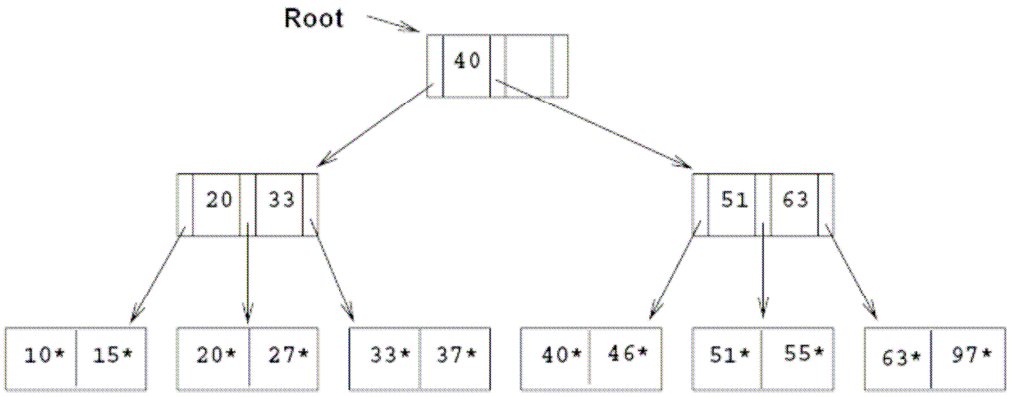
For inserts and deletes, the appropriate page is determined as for a search, and the record is inserted or deleted with overflow pages added if necessary.

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

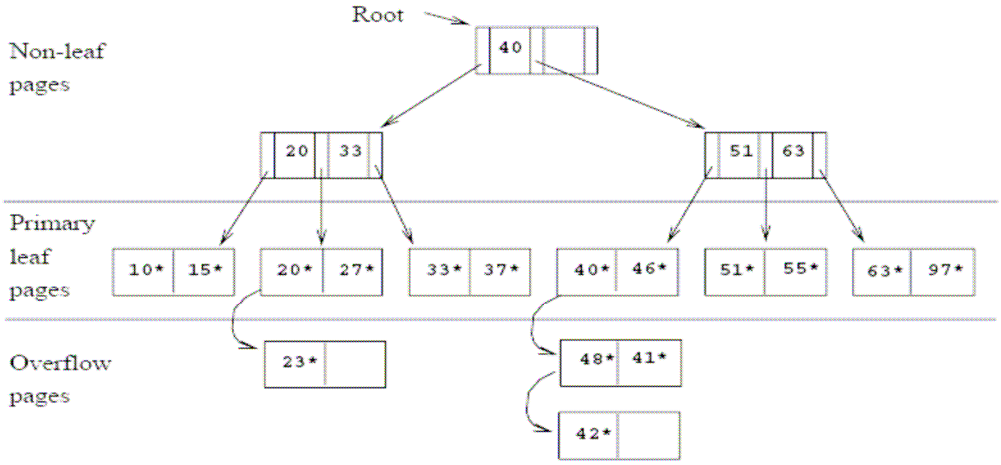
We assume that each leaf page can contain two entries. If we now insert a record with key value 23, the entry 23* belongs in the second data page, which already contains 20* and 27* and has no more space. We deal with this situation by adding an overflow page and putting 23* in the overflow page. Chains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

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

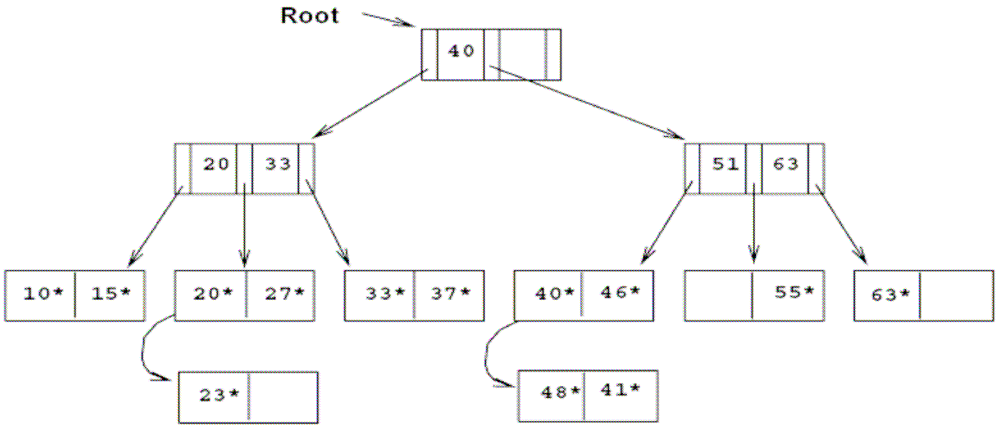
The non-leaf pages direct a search to the correct leaf page. The number of disk I/Os is equal to the number of levels of the tree and is equal to logf N, where N is the number of primary leaf pages and the **fan-out** F is the number of children per index page. This number is considerably less than the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



Sample ISAM Tree



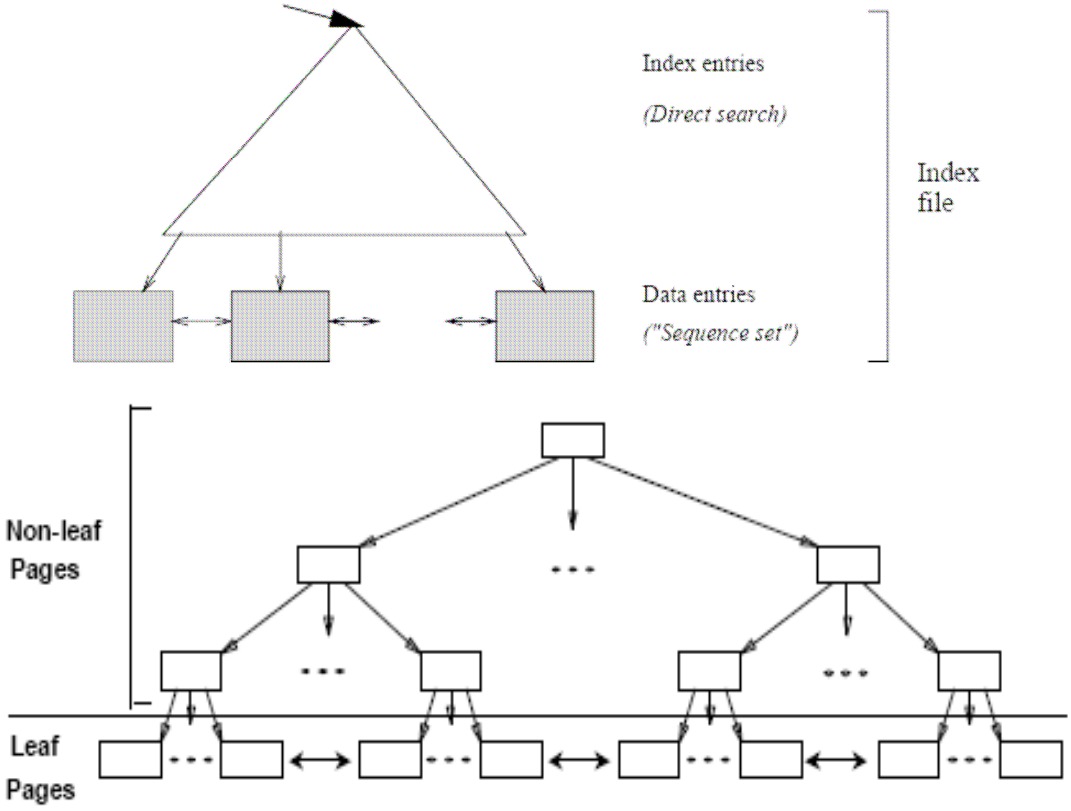
ISAM Tree after Inserts



ISAM Tree after Deletes

8.8 B+ Trees (A Dynamic Data Structure)

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



Structure of a B+ Tree

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

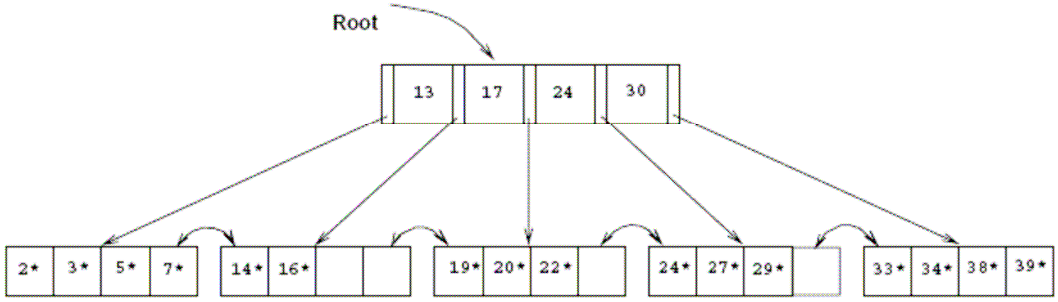
1. Operations (insert, delete) on the tree keep it balanced.
2. A minimum occupancy of 50 percent is guaranteed for each node except the root if the deletion algorithm is implemented. However, deletion is often implemented by simply locating the data entry and removing it, without adjusting the tree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

SEARCH:

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

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

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



INSERT:

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

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

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

The difference in handling leaf-level and index-level splits arises from the B+ tree requirement that all data entries kfi must reside in the leaves. This requirement prevents us from `pushing up' 5 and leads to the slight redundancy of having some key values appearing in the leaf

level as well as in some index level. However, range queries can be efficiently answered by just retrieving the sequence of leaf pages; the redundancy is a small price to pay for efficiency. In dealing with the index levels, we have more flexibility, and we `push up' 17 to avoid having two copies of 17 in the index levels.

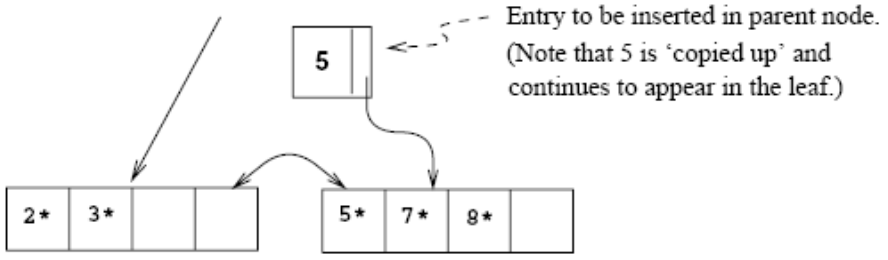


Figure 1 Split Leaf Pages during Insert of Entry 8*

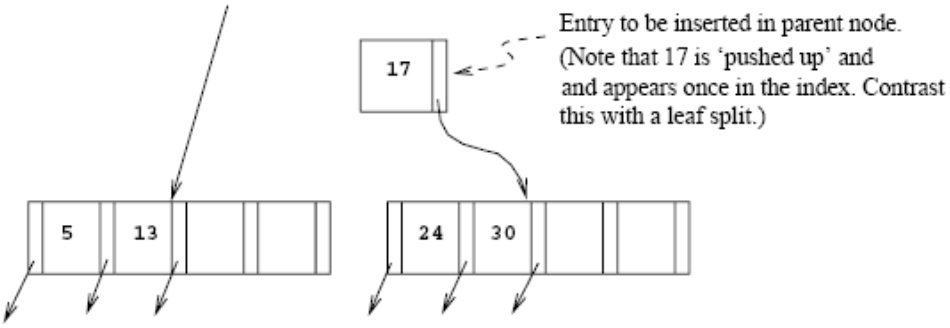


Figure 2 Split Index Pages during Insert of Entry 8*

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

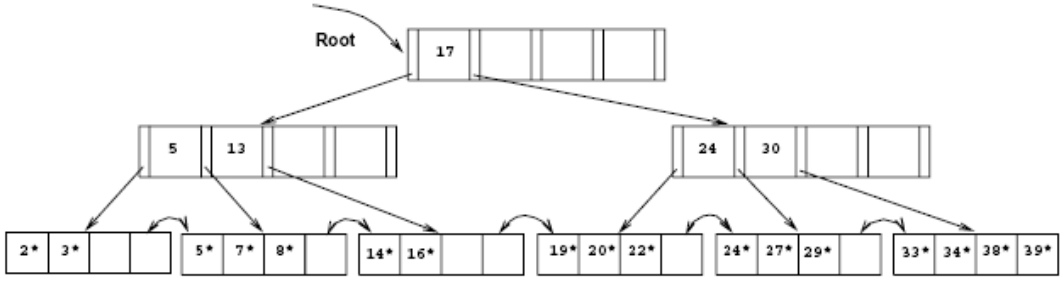


Figure 3 B+ Tree after Inserting Entry 8*

The entry belongs in the left-most leaf, which is full. However, the sibling of this leaf node contains only two entries and can thus accommodate more entries. We can therefore handle the insertion of 8* with a redistribution. Note how the entry in the parent node that points to the second

leaf has a new key value; we `copy up' the new low key value on the second leaf. This process is illustrated in Figure 9.4.

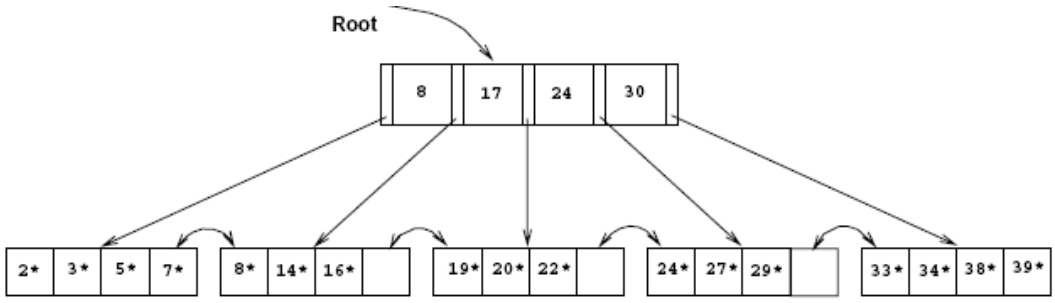


Figure 4 B+ Tree after Inserting Entry 8* Using Redistribution

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

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

DELETE:

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

returns to the parent node. If the last entry in the root node is deleted in this manner because one of its children was deleted, the height of the tree decreases by one.

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

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

Deleting the entry <27, pointer to second leaf page> has created a non-leaf-level page with just one entry, which is below the minimum of d=2. To fix this problem, we must either redistribute or merge. In either case we must fetch a sibling. The only sibling of this node contains just two entries (with key values 5 and 13), and so redistribution is not possible; we must therefore merge.

The situation when we have to merge two non-leaf nodes is exactly the opposite of the situation when we have to split a non-leaf node. We have to split a non-leaf node when it contains 2d keys and 2d + 1 pointers, and we have to add another key pointer pair. Since we resort to merging two non-leaf nodes only when we cannot redistribute entries between them, the two nodes must be minimally full; that is, each must contain d keys and d+1 pointers prior to the deletion. After merging the two nodes and removing the key pointer pair to be deleted, we have 2d−1 keys and 2d+1 pointers: Intuitively, the left-most pointer on the second merged node lacks a key value. To see what key value must be combined with this pointer to create a complete index entry, consider the parent of the two nodes being merged. The index entry pointing to one of the merged nodes must be deleted from the parent because the node is about to be discarded.

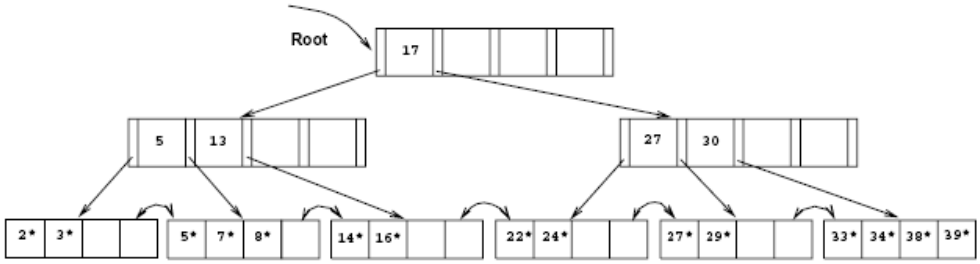


Figure 7 B+ Tree after Deleting Entries 19* and 20*

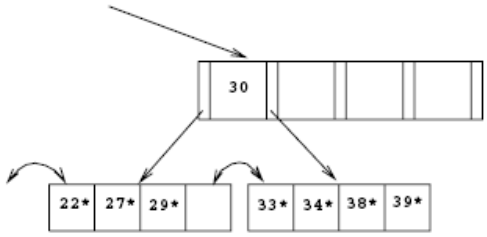


Figure 8 Partial B+ Tree during Deletion of Entry 24*

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

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

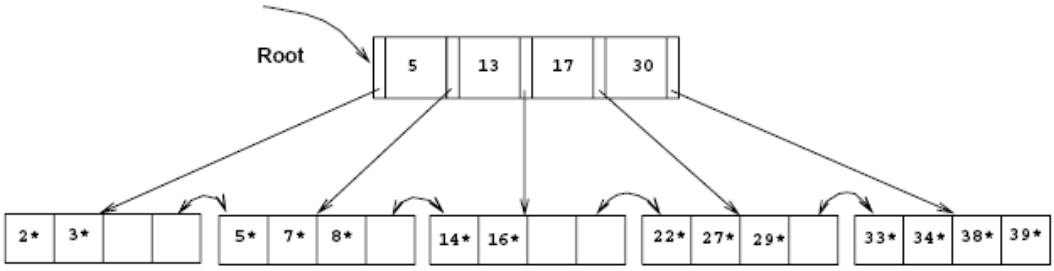


Figure 9 B+ Tree after Deleting Entry 24*

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

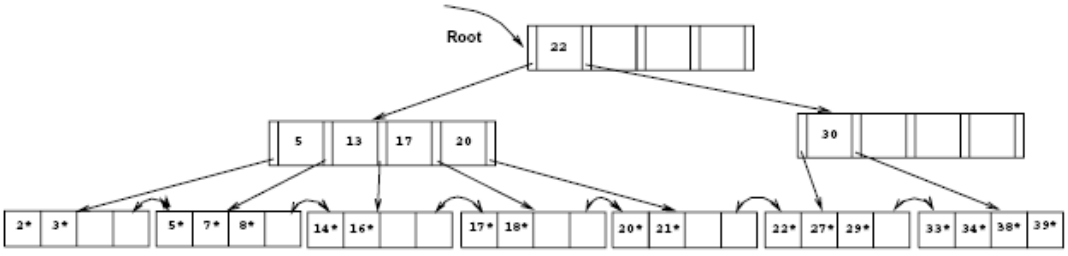


Figure 10 A B+ Tree during a Deletion

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

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

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

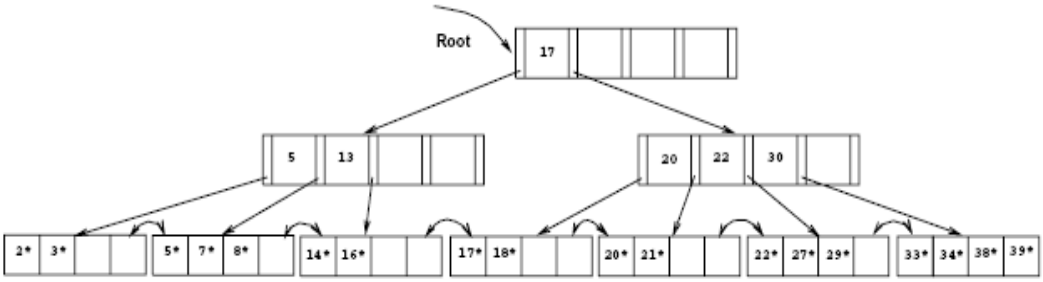


Figure 11 B+ Tree after Deletion

Transaction Management and Concurrency Control:

Transaction, properties of transactions, transaction log, and transaction management with SQL using commit rollback and save point.

Concurrency control for lost updates, uncommitted data, inconsistent retrievals and the scheduler. Concurrency control with locking methods : lock granularity, lock types, two phase locking for ensuring serializability, deadlocks, concurrency control with time stamp ordering : Wait/Die and Wound/Wait Schemes, Database Recovery management : Transaction Recovery.

Transaction Management: is the foundation for concurrent execution and recovery from system failure in DBMS

Transaction:

- Transaction is an execution of a user program.
- DBMS considers a transaction as a series of reads and writes.
- It performs a single unit of DB processing.
- Transactions have to be managed if the system is a multiuser system.

Properties of transaction:

A DBMS must ensure 4 important properties of transactions to maintain data in the face of concurrent access and system failure

They are

1. Atomicity
2. Consistency
3. Isolation
4. Durability

Atomicity :

- Atomicity means either all actions are carried out or none are.
- Users should not have to worry about the effect of incomplete transactions.
- Eg: Funds transfer (transfer of *Rs*. 500 from account A to account B)

  Read(A)
  A = A – 500
  Write(A)
  Read(B)
  B = B + 500
  Write(B)
- In the above example, either all operations should be done or none of the operations of transaction are done.
- Atomicity is maintained by the **Recovery Manager** (transaction management component of DBMS

Consistency : (correctness)

- Every user transaction must ensure that it will lead database instance from consistent state to another consistent state.
- Each transaction run by itself, with no concurrent execution of transactions, must preserve the consistency of database.
- The DBMS assumes that consistency holds for each transaction.
- Ensuring this property is the responsibility of the **user** (application developer).
- Eg: user can have consistency criterion like – fund transfer between bank accounts should not change the total amount of money in the account.

Isolation :

- In case of multiple transactions executing concurrently and trying to access a sharable resource at the same time, the system should create an ordering in their execution so that, they should not create any anomaly in values stored at the sharable resource.
- When multiple transactions execute simultaneously, the net result should be equal to executing the transactions in some serial order.
- This property is ensured by **concurrency control manager** component of DBMS.

Durability:

- If the transaction has been successfully completed, its effect should persist even if the system crashes before all its changes are reflected on disk. This property is called durability.
- Changes must not be lost due to some database failure. Changes should be permanent.
- This property is ensured by **Recovery manager** component of DBMS.

Discussion

- If each transaction maps a consistent database instance to another consistent database instance. Executing several transactions one after the other results in a consistent final database instance. User/application programmer is responsible for ensuring transaction consistency ? how?
  - By the logic of his program. By putting some criterion like total funds should be equal.
- The isolation property is ensured by guaranteeing that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions one after the other in some serial order.
- Transactions may be incomplete because of
  - They can be aborted / terminated unsuccessfully
  - System may crash
  - Transaction may encounter an unexpected situation.
- If a transaction is interleaved in the middle, it leaves the database in an inconsistent state.
- DBMS must ensure transaction atomicity. How to?
  - By undoing actions of incomplete transactions.
- How DBMS will undo?
  - With the help of log. DBMS maintains a log of all write to the database.
- Log is also used to ensure durability.
- Consider that a system crash happened. Log is used to remember and restore these changes when the system restarts.

Transaction Log

- The log is a history of actions executed by the DBMS. i.e., it keeps track of all transactions that update the database.
- Physically, the log is a file of records stored in stable storage, which is assumed to survive crashes.

- DBMS uses the information stored in a log for
  - *Recovery requirement triggered by a rollback statement.*
  - *A programs abnormal termination.*
  - *A system failure*

- Atomicity and Durability properties are ensured by the DBMS with the help of the log file.
    - *Atomicity property states that either all the operations of a transaction must be performed or none. The modifications done by an aborted transaction should not be visible to a database and the modifications done by a committed transaction should be visible. DBMS carries out undo or redo with the help of log file records.*
    - *Durability of log file can be achieved by maintaining two or more copies of the log on different disks, so that the chance of all copies of the log being lost is negligibly small.*

- Every log record is given a unique id called the **log sequence number** (LSN)
- **Log tail :** most recent portion of the log file, which is placed in main memory.
- Every log record has prevLSN, transID, and type. The set of all log records for a given transaction is maintained as a linked list going back in time, using the prevLSN field.

- A log record is written for each of the following actions.
    - ***Update :*** after modification, an update type record is appended to the log tail. Every update record contains transID, Dataitem that is being modified, old value and new values of the data item.
    - ***Start***: When a transaction starts, it is assigned a transaction id and that is written to the log file.
    - ***Commit :*** When a transaction decides to commit, it writes a commit type record containing the transaction id. That is the log record is appended to the log, and the log tail is written to stable storage, up to and including the commit record. The transaction is considered to have committed at the instant that its commit log record is written to stable storage.
    - ***Abort :*** When a transaction is aborted, an abort type log record containing the transaction is is appended to the log, and Undo is initiated for this transaction.

- With the help of the information stored in the log, system can perform redo and undo operations.
    - Undo – setting the data item to old value.
    - Redo – setting the data item to new value.

Recovery using log records :

After a system crash has occurred, the system consults the log to determine which transaction need to be redone and which need to be undone.

1. Transaction Ti needs to be undone if the log contains the record *<Ti start>* but does not contain either the record *<Ti commit>* or the record *<Ti abort>*
2. Transaction Ti needs to be redone if log contains record *<Ti start>* and either the record *<Ti commit>* or the record *<Ti abort>*

Transaction management with SQL

- In an abstract model of a transaction, a transaction is a sequence of reads, writes, and abort/commit actions.
- SQL provides support for users to specify transaction-level behaviour.

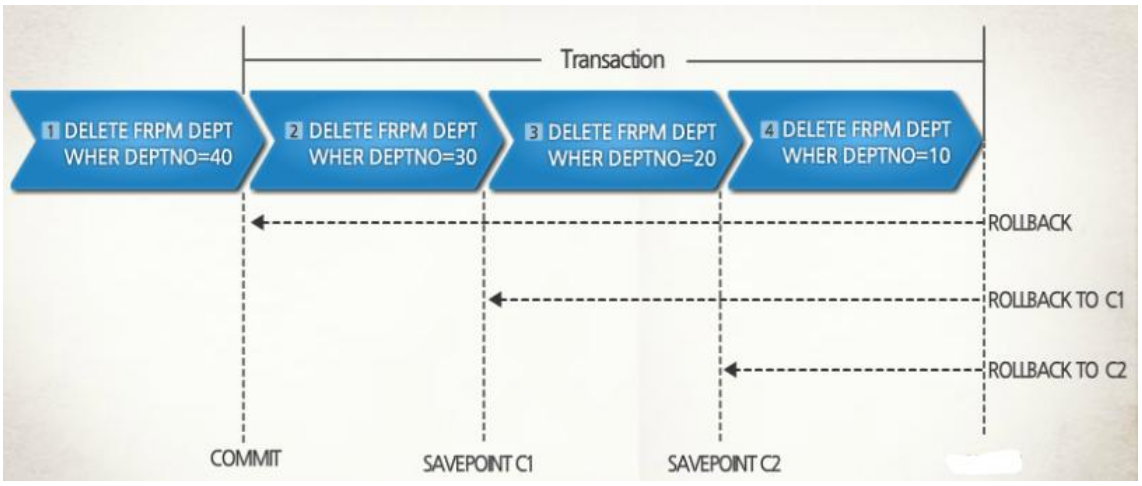
- SQL statements that provide transaction support are
    - **COMMIT**
        - Make changes done in transaction permanent.
        - Syntax : COMMIT;
    - **ROLLBACK**
        - It is an SQL keyword for abort.
        - Rollbacks the state of the database to the last commit point.
        - Syntax : ROLLBACK [to savepoint]
        - It can be used along with a savepoint to rollback upto the savepoint.
    - **SAVEPOINT**
        - Used to specify a point in transaction to which later you can rollback.
        - Syntax : SAVEPOINT name_of_savepoint

- These are considered as Transaction Control Language (**TCL**) of SQL.

- A transaction is automatically started when a user executes a statement that accesses either the database or the catalog, such as SELECT query, an UPDATE command, or a CREATE TABLE statement.
- Once a transaction is started, other statements can be executed as part of this transaction until the transaction is terminated by either COMMIT command or a ROLLBACK command.

- Transaction sequence must continue until
  - COMMIT statement is reached.
  - ROLLBACK statement is reached.
  - End of the program is reached.
  - Program is abnormally terminated.

- Example:



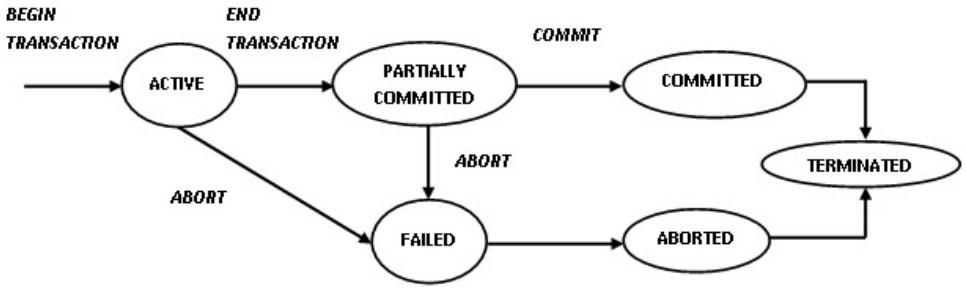
In the above example there are 4 queries.

Commit is executed after query 1, Savepoints C1 and C2 are executed after query 2 and 23.
Here a ROLLBACK undo all operations since commit, ROLLBACK to C1 will undo all operations since SAVEPOINT C1, and to C2 will undo all operations since SAVEPOINT C2.

Transaction States / Transaction State Diagram / Life Cycle of a Transaction

- A transaction goes through many different states throughout its lifetime. These states are known as transaction states.
- Transaction states are as follows-
  1. Active state
  2. Partially committed state
  3. Committed state
  4. Failed state
  5. Aborted state
  6. Terminated state

- The states of a transaction are illustrated in the following figure.



1. Active state-

- This is the first state in the lifecycle of a transaction.
- A transaction is called in an active state as long as its instructions are getting executed from first to last.
- All the changes made by the transaction are being stored in a buffer in main memory.

2. Partially committed state-

- After the last instruction of the transaction gets executed, it enters into a partially committed state.
- After the transaction has entered into this state, the transaction is now considered to be partially committed.
- It is not considered to be fully committed because all the changes made by the transaction are still stored only in the buffer in main memory and not into the database.

3. Committed state-

- After all the changes made by the transaction have been successfully stored into the database, the transaction enters into a committed state.
- NOTE- After a transaction has entered the committed state, it is not possible to roll back the transaction i.e. we can not undo the changes the transaction has made because the system has been now updated into a new consistent state.

4. Failed state-

- When a transaction is being executed or has partially committed and and some failure occurs due to some reason and it is analyzed that the normal execution is now impossible, the transaction then enters into a failed state.

5. Aborted state-

- After the transaction has failed and has entered into a failed state, all the changes made by the transaction have to be undone.
- To undo the changes made by the transaction, it is necessary to roll back the transaction.
- After the transaction has rolled back completely, it enters into an aborted state.

6. Terminated state-

- This is the last state in the life cycle of a transaction.
- After entering the committed state or aborted state, the transaction then finally enters into a terminated state where the transaction life cycle finally comes to an end.

Schedule

- A transaction is seen by the DBMS, as series, or list, of actions.
- The actions that can be executed by a transaction includes
  - reads and writes of db objects.
  - In addition each transaction must specify as its final action either commit or abort.

Denotion of actions of a transaction:

- ***$R_T(O)$*** : Read operation on database object O by transaction T
- ***$W_T(O)$*** : Write operation on database object O by transaction T
- ***$Abort_T$*** : Unsuccessful completion by transaction T (terminate and undo all the actions carried out thus far)
- ***$Commit_T$*** : Successful completion by transaction T (complete successfully)

- **Def :** A Schedule is a list of actions of several transactions.
- The order in which two actions of transaction T appear in a schedule must be the same as the order in which they appear in T.
- Following schedule shows an execution order for actions of 2 transactions T_1 and T_2.

T_1	T_2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	

- A schedule that contains either an abort or a commit for each transaction is called a ***complete schedule***.
- A schedule in which all actions of a transaction are executed from start to finish, one by one, without any interleaving is called ***serial schedule***.

Disadvantages of Serial Scheduling:

1. It limits concurrency.
2. It causes wastage of CPU time
3. Smaller transactions may need to wait long.

Because of this we will adopt non-serial scheduling.

Concurrent Execution of Transactions

- DBMS executes several transactions concurrently for performance reasons. i.e., DBMS interleaves the actions of different transactions.
- But not all interleaving should be allowed.
- DBMS should allow only safe interleaving of transactions (i.e., serializable schedules)

Motivations / reasons for concurrent execution :

- DBMS executes transactions concurrently for the following performance reasons.
  - Improved throughput
  - Improved response time / Reduced waiting time.
- The average number of transactions completed per unit amount of time is called system throughput. Concurrent execution of transactions improves system throughput. When one transaction is waiting for I/O, another transaction can be executed on CPU, (as CPU and I/O unit can work in parallel)
- Concurrent execution of transactions also improves response time, or the average waiting time taken to complete a transaction.
  Assume that in a serial execution of transactions, if a short transaction started after long transaction, short transaction could get stuck behind a long transaction. Interleaving of these two transactions usually allows the short transaction to complete quickly.

Serializability :

- A serializable schedule over a set **S** of committed transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S.
- i.e., the database instance that results from executing the given schedule is identical to the database instance that results from executing the transaction in some serial order.

Eg: A serializable schedule

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)
	Commit
Commit	

even though, the actions of T1 and T2 are interleaved, the result of this schedule is equal to running T1 and then running T2

Anomalies due to interleaved execution:

- DBMS interleaves the actions of several transactions for performance reasons, but not all interleaving are allowed.
- Two actions on the same data object conflict if at least one of them is a write.
- There are three main ways in which a schedule could run against a consistent database and leave it in an inconsistent state.
- Three anomalous situations due to interleaved execution of transactions are
    1. Write – Read conflict (WR)
    2. Read – Write Conflict (RW)
    3. Write – Write Conflict (WW)

1. Reading uncommitted Data (WR conflicts) :

- A transaction T2 could read a database object A that has been modified by another transaction T1 which has not yet been committed. Such a read is called 'Dirty Read'.
- ** **Dirty Read** – occurs when a transaction reads a database object that has been modified by another not yet committed transaction. **

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Schedule – 1

Reading uncommitted Data (Dirty Read)

The interleaved execution shown in schedule – 1 can lead to an inconsistency. Here, T2 performed a dirty read on database object A.

Eg: Let A = 500, B = 400
T1 – transfers $100 from A to B
T2 – increments both A and B by 10%.
when T1 and T2 runs alone, they preserve database consistency.

Execution of T1;T2 leads to a database state --> A= 440 B = 550

Execution of T2;T1 leads to a database state --> A= 450 B = 540

but interleaved execution leads to a database state --> A = 440, B = 540

This result is different from any result that we would get by running one of the two transactions first and then the other.
The problem is – The value of A written by T1 is read by T2 before T1 has completed all its changes.

T_1	T_2
R(A)	
A = A - 100	
W(A)	
	R(A)
	A = A * 1.1
	W(A)
	R(B)
	B = B * 1.1
	W(B)
	Commit
R(B)	
B = B + 100	
W(B)	
Commit	

2. Unrepeatable Reads (RW conflicts) :

- A Transaction T2 could change the value of an object A that has been read by a transaction T1, while T1 is still in progress.
- It T1 tries to read the value of A again, it will get a different result, even though it has not modified A in the mean time. It is called an unrepeatable read.
- This situation could not arise in serial execution of two transactions.

- Example : This example illustrates how this situation can lead to inconsistency.

T_1	T_2
R(A)	
A=A-1	
	R(A)
	A=A-1
	W(A)
	Commit
W(A)	
Commit	

Schedule-2 (unrepeatable read)

- Transactions are trying to book flight tickets.
- No.of tickets available (A) = 1
- T1 - read the value of A as 1 and proceeds to book the ticket.
- T2 – also reads the value of A as 1, booked the ticket (Written)
- T1 – trying to perform write operation.
- This leads to incorrect execution of transactions, there by inconsistency.

3. Overwriting uncommitted data (WW conflicts) :

- A transaction T2 could overwrite the value of an object A, which has been modified by a transaction T1, while T1 is still in progress.
- This is also called as Blind Write, because here the transaction writes to an object without ever reading the object.
- Example: There are two employees Motu and Patlu. Their salaries must be kept equal. Transaction T1 sets their salaries to $2000 and transaction T2 set their salaries to $1000

T_1	T_2
W(M)	
	W(M)
	W(P)
	Commit
W(P)	
Commit	

Schedule – 3(Blind Write)

- If we execute the transactions serially, then we will get both salaries as $1000 (T1;T2) or as $2000 (T2;T1)
- If we consider the interleaving shown in schedule-3, which consists of blind write leads to inconsistency.
- The above schedule makes Motu salary as 1000 and patlu salary as 2000.

Incorrect Summary problem:

- This **problem** is caused when one of the transactions is executing an aggregate operation on several data items, and other transactions are updating one or more of those data items. This causes a inconsistent database state.
- T2 is updating the value of C and is used in average calculation as one of the argument.

T_1	T_2
R(A)	
R(B)	
R(C)	
Sum=0	
Sum = sum + A	
	R(C)
	C=C+100
	W(C)
	Commit
Sum = sum + B	
Sum = sum + C	
Average()	
Commit	

Phantom Read and Phantom Tuple

When transaction T executes a query that retrieves a set of tuple from a relation satisfying a certain condition, reexecuting the query at a later time but find the retrieved set contains an additional (phantom) tuple that has been inserted by another transaction in the mean time. This is referred as phantom read.

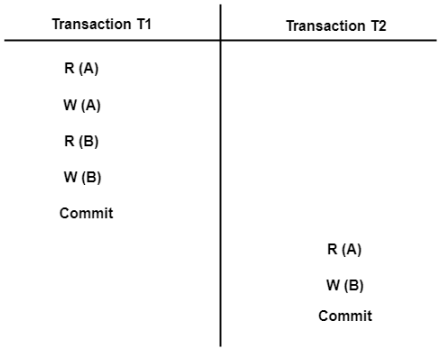
Schedule

- List of actions of several transactions as seen by the Database system is called as schedule.
- The various types of schedules are discussed below.

Serial Schedule:

- A schedule in which actions of different transactions are not interleaved.
- i.e., transactions are executed from start to finish, one by one.
- Serial schedules are always-
  - Consistent
  - Recoverable
  - Cascadeless
  - Strict

In the schedule shown in figure, we have two transactions T1 and T2 where transaction T1 executes first and after it complete its execution, transaction T2 begins its execution. So, this schedule is an example of a **serial schedule**.

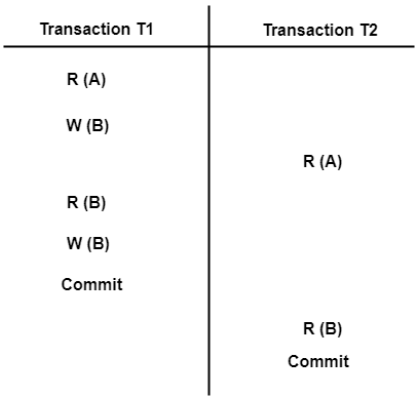


Serial Schedule

Non-Serial Schedule :

- Non-serial schedules are those schedules in which the operations of all transactions are interleaved or mixed with each other.
- Non-Serial schedules are **NOT** always-
  - Consistent
  - Recoverable
  - Cascadeless
  - Strict

In the schedule shown next, we have two transactions T1 and T2 where the operations of T1 and T2 are interleaved. So, this schedule is an example of a **non-serial schedule**.



Non-Serial Schedule

Complete Schedule:

A schedule that contains either an abort or commit for each transaction that listed in it.

T_1	T_2
R(A)	
	R(B)
	W(B)
	Commit
W(A)	
Commit	

Irrecoverable Schedule:

- If in a schedule, a transaction performs a dirty read operation (reads a value from an uncommitted transaction) and commits before the transaction from which it has read the value, then such a schedule is known as an **Irrecoverable schedule**.
- If T1 is aborted because of some reasons, then action of T1 will be rolled back. If T1 is rolled back, then T2 should also be rolled back, which is not possible here. This example schedule is not recoverable.

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
	Commit[1]
Abort[2]	

Irrecoverable Schedule

[1] A committed transaction should never be rolled back.
[2] All actions of aborted transactions are to be undone.

Recoverable Schedule:

Def: A schedule is recoverable if each transaction commits only after all transactions from which it has read has committed.

Or

A recoverable schedule is one where, for each pair of transactions T_i and T_j such that T_j reads data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j

- It is a schedule in which no committed transaction needs to be rolled back.
- If in a schedule, a transaction performs a dirty read operation (reads a value from an uncommitted transaction) and its commit operation is delayed till the time, the transaction from which it has read the value commits or roll back, then such a schedule is known as a **Recoverable schedule**.
- By delaying the commit operation of the transaction which has performed a dirty read, it is ensured that it still has a chance to roll back if the transaction which updated the value aborts or roll backs later.

T_1	T_2
R(X)	
W(X)	
	R(X)
	W(X)
Commit	
	Commit

Recoverable Schedule

Cascading abort:

Consider the following schedule

T_1	T_2	T_3
R(X)		
W(X)		
	R(X)	
	W(X)	
		R(X)
		W(X)
Abort		

- It T1 aborts, T2 also aborts, as it is reading data changed by T1 and T3 also aborts.
- It is a strictly avoidable case.
- It is called **cascading abort.**
- Due to cascading abort, cascading rollback happens.
- The phenomenon, in which a single transaction failure leads to a series of transaction rollbacks is called **cascading rollback**.
- Cascading rollback is undesirable, since it leads to the undoing of significant amount of work.

Cascadeless Schedule:

- It is a schedule which avoids cascading rollbacks.
- If transactions in a schedule read only the changes of committed transactions, then it is called cascadeless schedule.
- In cascadeless schedule, a write is allowed after write, but a read is allowed only after commit.

T_1	T_2	T_3
R(X)		
W(X)		
Commit		
	R(X)	
	W(X)	
	Commit	
		R(X)
		W(X)
		Commit

Cascadeless Schedule

Transaction T2 and T3 are reading X, after commit. i.e, T2 and T3 are reading a data item which is not dirty (safe)
This kind of schedule is called cascadeless schedule.

T_1	T_2
R(X)	
W(X)	
	W(X)
Commit	
	R(X)
	Commit

** Every cascadeless schedule is also recoverable schedule.

Strict Schedule:

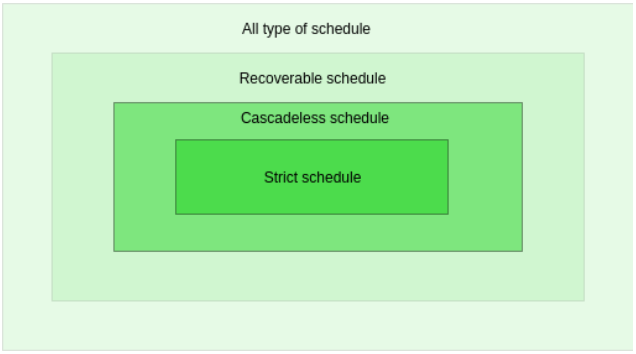
- A schedule is said to be strict if a value written by a transaction T is not read or overwritten by other transaction until T either aborts or commits.

T_1	T_2
R(X)	
	R(Y)
W(X)	
Commit	
	R(X)
	W(X)
	Commit

Strict Schedule

- Strict Schedules are recoverable, do not require cascading aborts, and actions of aborted transactions can be undone by restoring the original values of modified objects.
- Transaction T2 can neither perform read/write on X before T1 commits / aborts.
- If you are working on same data item, then strict schedule is serial schedule.
- If you are working with multiple data items strict schedules are not serial schedules.
- All strict schedules are cascadeless schedules.

Following figure illustrations the relation between Strict, cascadeless and recoverable schedules.



Serializable Schedule:

A serializable schedule over a set S of transactions is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over the set of Committed transactions in S.

Or

(in simple terms) A schedule whose result is equal to some serial schedule.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
R(B)	
W(B)	
	R(B)
	W(B)

Serializable Schedule

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	

Non-serializable Schedule

- Serial Scheduling has lot of problems in terms of performance.
- To overcome those, DBMS interleaves the execution of transactions (non-serial Scheduling)
- But some of the orders in non-serial scheduling will give incorrect results.
- Serializability helps us to check whether the order is correct or not. It takes the order, checks whether it is equal to some serial schedule, then we say that the schedule is serializable.
- A schedule **S** of **n** transactions is serializable if it is equivalent to some serial schedule of the same **n** transactions.
- Based on equivalence
  - Conflict Serializability
  - View Serializability

Conflict Serializable schedule:

- A schedule is conflict serializable if it is conflict equivalent to some serial schedule.
- Two schedules are said to be conflict equivalent if they involve the (same set of) actions of the same transactions, and they order every pair of conflict actions of two committed transactions in the same way.

Example 1: Determine whether the following 2 schedules are conflict equivalent or not.

T_1	T_2
R(A)	
	R(A)
	R(E)
R(B)	
R(C)	
W(C)	
	R(C)
W(D)	
R(E)	
	R(D)
	R(F)
	W(F)

T_1	T_2
R(A)	
R(B)	
	R(A)
R(C)	
W(C)	
	R(E)
	R(C)
W(D)	
	R(D)
	R(F)
	W(F)
R(E)	

Answer : Both Schedules are conflict equivalent, as they have the same set of operations in both schedules, they order the conflicting operations in the same order.

Example 2: Determine whether the given schedule is conflict serializable or not.

T_1	T_2
R(A)	
	R(A)
	R(E)
R(B)	
R(C)	
W(C)	
	R(C)
W(D)	
R(E)	
	R(D)
	R(F)
	W(F)

Answer: we can say that the given schedule is conflict serializable, if it is conflict equivalent to some serial schedule. The given schedule is conflict equivalent to the serial schedule <T1, T2>. Therefore the given schedule is conflict serializable

Example 3: Check whether the given schedule is conflict serializable or not.

S: $R_1(X)$ $R_2(X)$ $W_1(X)$ $R_1(Y)$ $W_2(X)$ $W_1(Y)$

Answer :

It is hard and time consuming to check whether the given schedule is conflict serializable or not in this way.
There is another method, in which we can construct precedence graph & check for cycles. If the precedence graph is acyclic, we can say that given schedule is conflict serializable.

Precedence Graph (Serializability Graph):

- Precedence graph is the best method to test for conflict Serializability.
- Precedence graph (or Serializability graph) is useful to capture all potential conflict between the transactions in a schedule.
- The precedence graph for a schedule S contains
  - A node for each committed transaction S
  - An arc from T_i to T_j if an action of T_i precedes and conflict with one of the T_j's actions

****if the precedence graph (of a given schedule) is acyclic, we can say that the given schedule is conflict serializable.
**No.of nodes in a precedence graph = no.of transactions in a schedule.

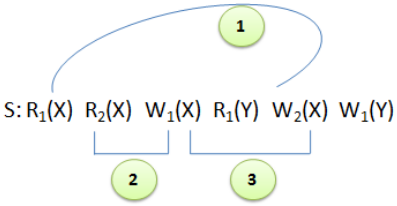
Example : Check whether the following Schedule is conflict serializable or not.

S: $R_1(X)$ $R_2(X)$ $W_1(X)$ $R_1(Y)$ $W_2(X)$ $W_1(Y)$

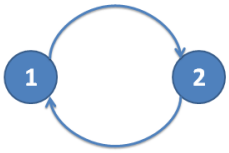
There are 3 conflicting operations.

R1(X) W2(X)
R2(X) W1(X)
W1(X) W2(X)

They were shown in the following picture.



Precedence graph for the above conflict operations is



The precedence graph is cyclic, therefore the given schedule is not conflict serializable.

Check yourself.

1. Check whether the given schedule is conflict serializable or not.
   S: R2(Z) R2(Y) W2(Y) R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z)
2. Check whether the given schedule is conflict serializable or not.
   S: R2(Z) R2(Y) W2(Y) R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z) R2(X) R1(Y) W1(Y) W2(X)

View Serializable Schedule:

- A schedule is called view serializable if it is view equivalent to some serial schedule.

- View equivalence: Two schedules S1 and S2 over the same set of transactions are said to be view equivalent, iff the following conditions are satisfied.
  1. *Initial read :* if Ti reads the initial value of object A in S1, then it must also read the initial value of A in S2.
  2. *Updated read :* If Ti reads a value of A written by Tj in S1, it must also read the value of A written by Tj in S2.
  3. *Final Write operation :* For each data object A, the transaction (if any) that performs the final write on A in S1 must also perform the final write on A in S2.

Note:

- If the given schedule is conflict serializable, then it is surely view serializable.
- If the given schedule is not conflict serializable, then it may or maynot be view serializable.
- No blind write means, not a view serializable schedule.

Example : Check whether the given two schedules are view equivalent or not.

T_1	T_2	T_3
R(A)		
	W(A)	
	Commit	
W(A)		
Commit		
		W(A)
		Commit

Schedule - 1

T_1	T_2	T_3
R(A)		
W(A)		
Commit		
	W(A)	
	Commit	
		W(A)
		Commit

Schedule – 2

These two schedules are view equivalent to each other.
Schedule-1 is view serializable, as it is view equivalent to the serial schedule-2

Check your understanding:

1. Determine whether the following schedule is (1) conflict serializable (2) view serializable.

T_1	T_2
R(X)	
	R(X)
W(X)	
	W(X)

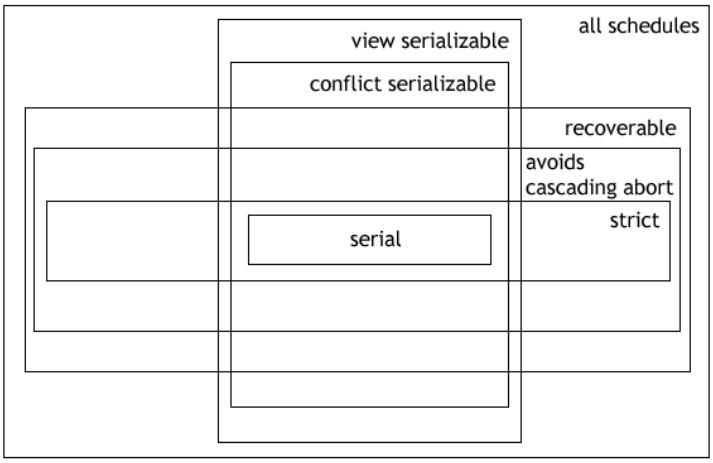
2. Determine whether the following schedule is (1) conflict serializable (2) view serializable

T_1	T_2	T_3
	R(B)	
	W(A)	
R(A)		
		R(A)
W(B)		
	W(B)	
		W(B)

Important points :

- Every conflict serializable schedule is view serializable, but every view serializable schedule need not be conflict serializable.
- Strict 2PL allows only conflict serializable schedules.
- Schedule which is view serializable, but not conflict serializable will have Blind Writes.

The relationship among various schedules is shown below.



Concurrency Control

- Concurrency control is the simultaneous execution of transactions to ensure serializability in multiuser database.
- DBMS executes several transactions concurrently for performance reasons. All interleaving are not safe. DBMS should allow only safe interleavings.
- DBMS must be able to ensure important properties of schedules like serializability and recoverability.
- To ensure these DBMS uses several concurrency control mechanisms, some of them are
    - Lock based protocols,
    - Time-stamp based protocols.

Lock-based concurrency control

- It is the most widely used concurrency control mechanism by DBMS.
- Locking protocol is typically used by DBMS to ensure only serializable and recoverable schedules.
- A lock is a small book keeping object associated with a database object.

- Locking is a procedure used to control concurrent access to data (to ensure serializability of concurrent transactions)
- In order to use a 'resource' (table, row, etc) a transaction must first acquire a lock on that resource
- This may deny access to other transactions to prevent incorrect results

- A **locking protocol** is a set of rules to be followed by each transaction to ensure that, even thought actions of several transactions might be interleaved; the net effect is identical to executing all transactions in some serial order.

Lock Granularity:

- The size of the database object being locked is called lock granularity. It can be coarse or fine.
- If a small database object is locked, then that is referred as fine level granularity.
- If a large database object is locked, then that is referred as coarse level granularity.
- Finer granularity improves concurrency.
- Lock granularity indicates the level of access to a database defined by a lock manager.
- There are five different levels of access in locking granularity. They are

1. **Database level:**
   Only one transaction is allowed to access the entire database at a time.
   It is coarse level of locking.
   It ensures serializability, but decreases concurrency.
2. **Table level**
   Each table allows only one transaction to interact at a time.
   Other transactions are allowed to access other tables in the database.
   It is less restricted than database level locking.
3. **Page level**
   A transaction can lock only stipulated number of rows from one or more tables.
   A page is a virtual table on one or many tables.
4. **Row level**
   Row level locking allows more than one transaction on a table as long as their rows are different.
   Each locked row allows one transaction to interact with it at a time.
   It is less restrictive than the previous ones.
5. **Field level**
   This locking allows concurrency control to put locking on fields.
   Each locked attribute allows only one transaction to interact with at a time.
   It improves concurrency.
   It is most flexible and least restrictive.

Types of locks:

- Any transaction cannot read or write data until it acquires an appropriate lock on it.
- The various types of locks are

1. **Shared lock / Read Lock (S)** : If a transaction wants to perform read operation on a database object, first it has to acquire shared lock on that object.

2. **Exclusive lock / Write Lock (X) :** If a transaction wants to perform write operation on a database object, first it has to acquire exclusive lock on it.

The DBMS grants locks to transactions according to the following table when a transaction request the DBMS for locks.

State of the lock \ Lock request type	Shared	Exclusive
Shared	Yes	No
Exclusive	No	No

Lock compatibility matrix

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions.
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive lock on the item, no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. Then the lock is granted.

Lock Management:

- **Lock Manager :**
  - The part of the DBMS that keeps track of the locks issued to transactions.
  - It maintains a lock table.
- **Lock Table:**
  - It is a hash table with the data object identifier as the key.
  - A lock table entry for an object contains
    - The number of transactions currently holding a lock on the object,
    - The nature of the lock (shared or exclusive) and
    - A pointer to a queue of lock requests.
- **Transaction table:**
  - It contains details of transactions.
  - Every entry contains a pointer to a list of locks held by the transaction.
  - This list is checked before requesting a lock, to ensure that a transaction does not request the same lock again.

- When a transaction needs a lock on an object, it issues a lock request to the lock manager.
  - If a shared lock is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object.
  - If an exclusive lock is requested and no transaction currently holds a lock on the object, the lock manager grants the lock and updates the lock table entry.
  - Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object.

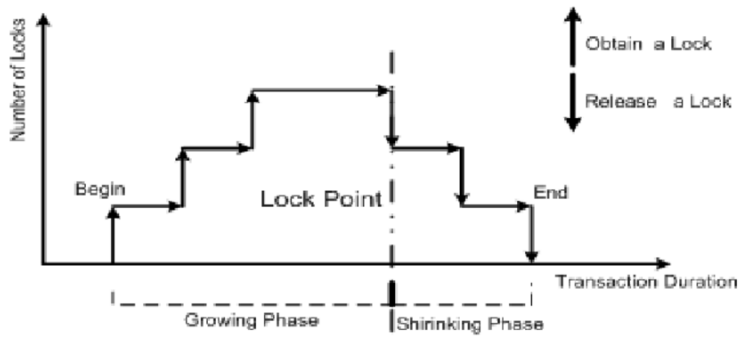
- When a transaction aborts or commits, it releases all its locks.

Two Phase Locking Protocol (2PL):

2PL for concurrency control:

- A transaction is said to follow two phase locking protocol if locking and unlocking can be done in two phases.

- **Growing phase:** new locks on data items can be acquired but none can be released.
- **Shrinking Phase:** existing locks may be released but no new locks can be acquired.



- **Lock point** is the point at which the growing ends (i.e., when the transaction takes final lock it needs to carry out its work)

- 2PL ensures serializability. If every transaction in a schedule follows the two-phase locking protocol, the schedule is guaranteed to be serializable.

Skeleton transaction implementing 2-PL

	T_1	T_2
1	LOCK-S(A)	
2		LOCK-S(A)
3	LOCK-X(B)	
4		
5	UNLOCK(A)	
6		LOCK-X(C)
7	UNLOCK(B)	
8		UNLOCK(A)
9		UNLOCK(C)

Transaction T1:
Growing Phase : steps 1 to 3
Shrinking Phase : steps 5 to 7
Lock Point : step 3

Transaction T2:
Growing Phase : steps 2 to 6
Shrinking Phase : steps 8 to 9
Lock Point : step 6

- Drawbacks of 2PL are
  - Unnecessary wait due to early locks
  - Deadlocks and starvation is possible.
  - Cascading rollback is possible in 2PL.

1) <u>Unnecessary locks due to early locks</u>

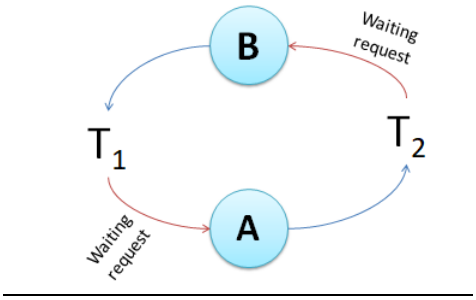
T_1	T_2
L(A)	
L(B)	L(A)
L(C)	

In the above schedule, lock point is where it locked C. From lock point onwards unlocking starts.
Assume that T1 completes its work on data object A, at the very beginning. Even though T2 has to wait for A until shrinking phase of T1 starts and T1 releases A.

2) <u>Deadlock:</u>

- Transaction T1 acquired an exclusive lock on data object B.
- Transaction T2 acquired a shared lock on data object A.
- Transaction T2 requested an exclusive lock on object B, which can't be granted until T1 releases the lock. So it is waiting.
- Transaction T1 requested an exclusive lock on object A, which can't be granted until T2 releases the lock. So it is waiting.
- This is a deadlock.

T_1	T_2
Lock-X(B)	
R(B)	
W(B)	
	Lock-S(A)
	R(A)
	Lock-X(B)
Lock-X(A)	



Therefore, deadlocks may come in 2PL

3) <u>Cascading rollback:</u>
Consider the following schedule. If T1 aborts, then it rollbacks the actions of T2 and T3. Therefore cascading rollback happens in 2PL.

T_1	T_2	T_3
LOCK-X(A) R(A) W(A) UNLOCK(A)		
	LOCK-X(A) R(A) W(A) UNLOCK(A)	
		LOCK-X(A) R(A)
abort		

Following are the variants which improves the 2PL

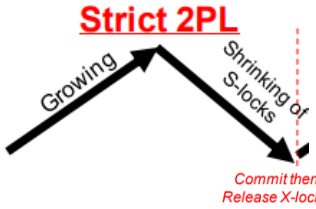
1. Strict 2PL
2. Rigorous 2PL
3. Conservative 2PL

Conservative 2PL:

- This locking protocol acquires locks on all the desired data items before transaction begins its execution.
- There is no growing phase.
- It guarantees serializability.
- It will not guarantee strict schedule, as some transaction can make a dirty read from T, before T has committed.
- It is deadlock free, as it acquires all locks before starting its execution.
- It can have cascading rollback problem.
- Practical implementation of this protocol is difficult, as it needs early prediction.

Strict 2PL:

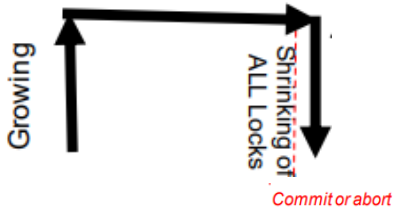
- In this protocol
  - Exclusive locks are only released after commit or abort.
  - Shared locks are released earlier.



- It is practically possible, in fact It is most commonly used 2PL algorithm.
- It guarantees serializability, but it is not deadlock-free.
- As dirty-reads are not allowed, it ensures strict schedules for recoverability.
- It guarantees strict schedules.

Rigorous 2PL:

- It is even more restrictive than strict 2PL.
- In this protocol, exclusive and shared locks are only released after commit or abort.



- It is deadlock free
- It gurantees serializability
- Leads to strict schedules for recoverability.

Problems of locking:

- Problems that may occur when we use locking
    1. Deadlock
    2. Starvation

Deadlock:

- Deadlock occurs when each transaction T in a set of two or more transactions is waiting for some item that is locked by some other transaction T^1 in the set.
- Hence each transaction in the set is in a waiting queue, waiting for one of the other transactions in the set to release the lock on an item. But because the other transaction is also waiting, it will never release the lock.
- To prevent deadlocks the time stamp concept is used.

Transaction Timestamp TS(t)

- It is an increasing variable (integer) indicating the age of an a transaction.
- It is a unique identifier assigned to each transaction
- The timestamps are typically based on the time at which the transaction is started
- A larger timestamp indicates a more recent transaction
- If TS(t1) < TS(t2) then t2 is younger than t1
- Two schemes to prevent deadlocks based on timestamps are
    1. Wait – Die
    2. Wait – Wound

Wait – Die

- Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rule followed by Wait – Die scheme is

If TS(Ti) < TS(Tj) then
Ti is allowed to wait;
Otherwise
Abort Ti and restart it later with the same timestamp.

Wound – Wait

- Suppose that transaction Ti tries to lock an item X but is not able to because X is locked by some other transaction Tj with a conflicting lock. The rule followed by Wound – Wait scheme is

If TS(Ti) < TS(Tj) then
abort Tj and restart it later with the same timestamp;
Otherwise
Ti is allowed to wait.

Starvation

- Occurs when a transaction cannot proceed for an indefinite period of time while other transactions in the system continue normally.
- It occurs if the waiting scheme for locked items is unfair.
- Solutions for starvation:
    - **A fair waiting scheme (FCFS**) – transactions are enabled to lock an item in the order in which they originally requested the lock.
    - **Aging** – Allowing transactions to have priority, and increasing the priority of the transactions the longer it waits.

Concurrency control based on timestamp ordering

What is a Timestamp?

- It is a unique identifier created by the DBMS to identify a transaction.
- TS values are assigned in the order in which the transactions are submitted to the system.
- A timestamp can be thought of as the transaction start time.
- TS(t) – refers to timestamp of T
- If TS(t1) < TS(t2) then t2 is younger than t1

- Concurrency control techniques based on timestamps do not use locks; hence, deadlocks cannot occur.

Timestamps are assigned in two ways

1. **Using current date and time** : The current date or time value of the system clock can be used as timestamp. Here, it ensures that no two timestamp values are generated during the same tick of the clock.
2. **Using a logical counter** : A counter can be used which is incremented each time its value is assigned to a transaction. Transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to 0.

Basic TimeStamp Ordering Alogirthm:

- This algorithm based on timestamps and ensure serializability using the ordering of timestamps
- **Idea** – to enforce the equivalent serial order on the transactions based on their timestamps. Transactions are ordered on the basis of arrival time.
- The algorithm allows interleaving of transaction operations, but it must ensure that for each pair of conflicting operations in the schedule, the order in which the item is accessed must follow the Timestamp order.

- To do this, the algorithm associates with each database item X, two timestamp values
    1. Read time stamp of X RTS(X)
    2. Write time stamp of X WTS(X)
- **RTS(X)** – the read timestamp of item X is the largest timestamp among all timestamps of transactions that have successfully read item X.
- **WTS(X)** – the write timestamp of item X is the largest of all the timestamps of transactions that have successfully written item X.

This algorithm uses the following assumptions:

- A data item X in the database has a RTS(X) and WTS(X) (recorded when the object was last accessed for the given action)
- A transaction T attempts to perform some action (read or write) on data item X on timestamp TS(T)
- Problem: We need to decide whether T has to be aborted or whether T can continue execution.

- This algorithm utilizes Timestamps to guarantee serializability of concurrent transactions.

Time Stamp Ordering Rule :

> If Pi(X) and Qj(X) are conflicting operations then
> Pi(X) is processed before Qj(X) iff **TS(Ti) < TS(Tj)**

- If the TO rule is enforced in a schedule then the schedule is (conflict) serializable.
- Why? Because cycles are not possible in the Conflict Precedence Graph

Algorithm:

Whenever some transaction T tries to issue R(X) or W(X) operation, it compares TS(T) with RTS(X) and WTS(X) to ensure that the TO of transaction execution is not violated.

Case 1: Whenever a transaction T issues W(X) operation

If RTS(X) > TS(T) or if WTS(X) > TS(T) then
Abort and rollback T and reject the operation
Otherwise
Execute W(X) operation of T and
set WTS(X) = TS(T)

Case 2: Whenever a transaction T issues R(X) operation

If WTS(X) > TS(T) then
Abort and rollback T and reject the operation
If WTS(X) <= TS(T) then
Execute R(X) operation of T and
Set RTS(X) = largest(RTS(X), TS(T)

Advantages:

- Schedules are serializable (like 2PL protocols) : Whenever this algorithm detects two conflicting operations, that occur in the incorrect order, it rejects the later of the two operations by aborting the transaction that issued it. Hence, the schedules produced by basic TO are guaranteed to be conflict serializable.
- No waiting for transaction, thus, no deadlocks!

Disadvantages:

- Schedule may not be recoverable (read uncommitted data)
  - Solution: Utilize Strict TO Algorithm (see next)
- Starvation is possible (if the same transaction is continually aborted and restarted)
  - Solution: Assign new timestamp for aborted transaction

May 2018

1. Define transaction and explain desirable properties of transactions.
2. Compare wait/die with wound/wait scheme.

November 2015

1. Illustrate lost update problem with suitable example.
2. Draw transaction state diagram and describe each state that a transaction goes through during its execution.
3. Explain in detail about timestamp based concurrency control techniques.
4. Briefly discuss about different types of schedules.
5. Discuss about different types of failures.
6. What is 2-phase locking protocol? How does it guarantee serializability?
7. Describe Wait/Die & Wound/Wait protocols.
8. Why the concurrency control is needed? Explain it. [8M]
9. Write and explain optimistic concurrency control algorithm.
10. Write short notes on:
    i) Phantom Record ii) Repeatable Read iii) Incorrect Summary iv) Dirty Read
11. Describe Wait/Die and Wound/Wait deadlock protocols.
12. Explain WAL protocol.

May 2016

1. Explain about deadlocks.
2. Explain the time stamp based protocol for concurrency control in a DBMS. [8M]
3. Explain the ARIES recovery method. When does a system recover from a crash? In what order must a transaction be undone and redone? Why is this order important?

May 2017

1. Define the term ACID properties.
2. Explain read-only, write-only and read-before-write protocols in serializability. [8M]
3. How the use of 2PL would prevent interference between two transactions.

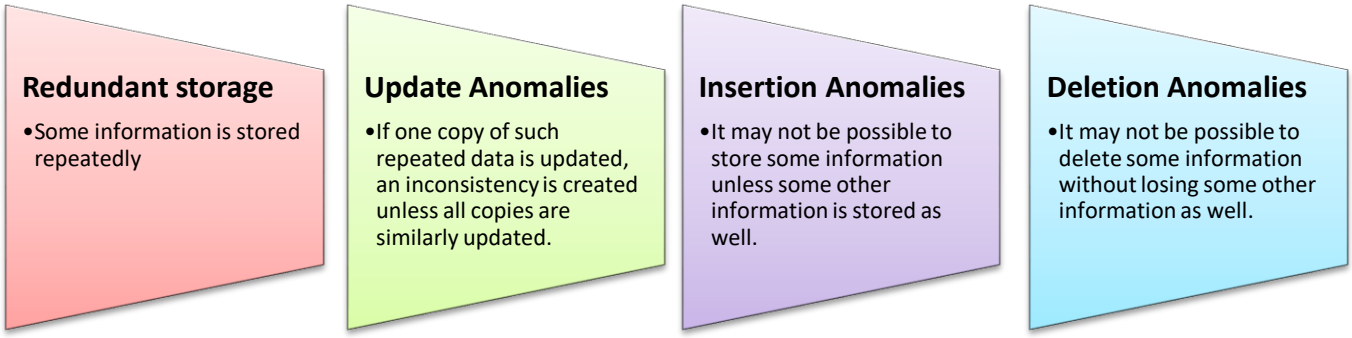
November 2016

1. State and explain two-phase locking protocol.
2. What is transaction? Mention the desirable properties of a transaction. [6M]
3. Discuss about transaction recovery techniques. [10M]
4. What is transaction log? Mention its content.
5. Why concurrency control is needed? Explain the problems that would arise when concurrency control is not provided by the database system.
6. What is serialization? Explain it.
7. Write about the transaction management with SQ L using commit, rollback, and savepoint.
8. Briefly discuss about various lock based mechanisms used in concurrency

Schema Refinement (Normalization) – Purpose of Normalization or schema refinement, concept of functional dependency, normal forms based on functional dependency (1NF, 2NF and 3NF), concept of surrogate key, Boyce - codd normal form (BCNF), Lossless join and dependency preserving decomposition, Fourth normal form (4NF).

SCHEMA REFINEMENT

- The Schema Refinement refers to refine the schema by using some technique. The best technique of schema refinement is decomposition.
- Normalization or Schema Refinement is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies. Redundancy refers to repetition of same data or duplicate copies of same data stored in different locations.
- Anomalies: Anomalies refers to the problems occurred after poorly planned and normalised databases where all the data is stored in one table which is sometimes called a flat file database.
- Problems caused by redundancy
  - ✓ **Redundant storage**
  - ✓ **Update Anomalies**
  - ✓ **Insertion Anomalies**
  - ✓ **Deletion Anomalies**



Consider the relation

Hourly_emps (ssn, name, lot, rating, hourly_wages, hours_worked)

The key for Hourly Emps is ssn. In addition, suppose that the hourly wages attribute is determined by the rating attribute. That is, for a given rating value, there is only one permissible hourly wages value. This IC is an example of a functional dependency. It leads to possible redundancy in the relation Hourly Emps

ssn	name	lot	Rating	Hourly_wages	Hours_worked
123-22-3666	Attishoo	48	8	10	40
231-31-5368	Smiley	22	8	10	30
131-24-3650	Smethurst	35	5	7	30
434-26-3751	Guldu	35	5	7	32
612-67-4134	Madayan	35	8	10	40

If the same value appears in the rating column of two tuples, the IC tells us that the same value must appear in the hourly wages column as well. This redundancy has several negative consequences:

- Some information is stored multiple times. For example, the rating value 8 corresponds to the hourly wage 10, and this association is repeated three times. In addition to wasting space by storing the same information many times,redundancy leads to potential inconsistency.
- For example, the *hourly wages* in the first tuple could be updated without making a similar change in the second tuple, which is an example of an ***update anomaly*.** Also, we cannot insert a tuple for an employee unless we know the hourly wage for the employee's rating value, which is an example of an *insertion anomaly*.
- If we delete all tuples with a given rating value (e.g., we delete the tuples for Smethurst and Guldu) we lose the association between that rating value and its hourly wage value (a **deletion anomaly**).

Null Values:

- ✓ null values cannot help eliminate redundant storage or update anomalies.
- ✓ null values can address insertion and deletion anomalies
- ✓ insertion anomaly example
  - we can insert an employee tuple with null values in the hourly wage field. null values cannot address all insertion anomalies. For example, we cannot record the hourly wage for a rating unless there is an employee with that rating
- ✓ deletion anomaly example
  - we might consider storing a tuple with null values in all fields except rating and hourly wages if the last tuple with a given rating would otherwise be deleted. However, this solution will not work because it requires the ssn value to be null, and primary key fields cannot be null.
- ✓ • Null values do not provide a general solution to the problems of redundancy,

Decomposition

- Technique used to eliminate the problems caused by redundancy.
- Def: A decomposition of a relation schema R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes of R.
- in simple words, Splitting the relation into two or more sub tables

(or)

The essential idea is that many problems arising from redundancy can be addressed by replacing a relation with a collection of `smaller' relations. Each of the smaller relations contains a (strict) subset of the attributes of the original relation. We refer to this process as *decomposition* of the larger relation into the smaller relations.
For example:

We can deal with the redundancy in Hourly Emps by decomposing it into two relations:
Hourly Emps2(ssn, name, lot, rating, hours worked)
Wages(rating, hourly wages)

ssn	name	lot	Rating	Hours_worked
123-22-3666	Attishoo	48	8	40
231-31-5368	Smiley	22	8	30
131-24-3650	Smethurst	35	5	30
434-26-3751	Guldu	35	5	32
612-67-4134	Madayan	35	8	40

Rating	Hourly_wages
8	10
5	7

Note that we can easily record the hourly wage for any rating simply by adding a tuple to Wages, even if no employee with that rating appears in the current instance of Hourly Emps. Changing the wage associated with a rating involves updating a single Wages tuple. This is more efficient than updating several tuples (as in the original design), and it also eliminates the potential for inconsistency. Notice that the insertion and deletion anomalies have also been eliminated.

Problems Related to Decomposition
Unless we are careful, decomposing a relation schema can create more problems than it solves. Two important questions must be asked repeatedly:

1. Do we need to decompose a relation?
2. What problems (if any) does a given decomposition cause?

To help with the first question, several *normal forms* have been proposed for relations. If a relation schema is in one of these normal forms, we know that certain kinds of problems cannot arise. Considering the normal form of a given relation schema can help us to decide whether or not to decompose it further. If we decide that a relation schema must be decomposed further, we must choose a particular decomposition (i.e., a particular collection of smaller relations to replace the given relation).

With respect to the second question, two properties of decompositions are of particular interest.
The ***lossless-join* property** enables us to recover any instance of the decomposed relation from corresponding instances of the smaller relations.
The ***dependency-preservation* property** enables us to enforce any constraint on the original relation by simply enforcing some constraints on each of the smaller relations. That is, we need not perform joins of the smaller relations to check whether a constraint on the original relation is violated.

FUNCTIONAL DEPENDENCIES

A functional dependency (FD) is a kind of Integrity constraint that generalizes the concept of a key.
Let R be a relation schema and let X and Y be nonempty sets of attributes in R. We say that an instance r of R satisfies the Functional Dependency **X → Y**
if the following holds for every pair of tuples t1 and t2 in r:

If t1:X = t2:X, then t1:Y = t2:Y .

We use the notation t1:X to refer to the projection of tuple t1 onto the attributes in X, in a natural extension of our TRC notation t:a for referring to attribute a of tuple t.
An FD X -> Y essentially says that if two tuples agree on the values in attributes X, they must also agree on the values in attributes Y.

Example:

Table Student

studID	lastname	status	credits
1234567	Smith	Undergraduate	254
1234568	Don	Undergraduate	149
1234569	Dallas	Graduate	543

Functional dependencies include:

{stuId}→{lastName}
{stuId} →{lastName, credits, status, stuId}
{credits}→{status} (but not {status}→{credits})

Example :

A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a2	b1	c3	d1

Instance r of R

AB → C	Holds on r
B → C	Doesn't Hold on r
A → B	Doesn't Hold on r
ABC → D	Doesn't Hold on r
CD → A	Holds on r

- A primary key constraint is a special case of an FD. The attributes in the key play the role of X, and the set of all attributes in the relation plays the role of Y.

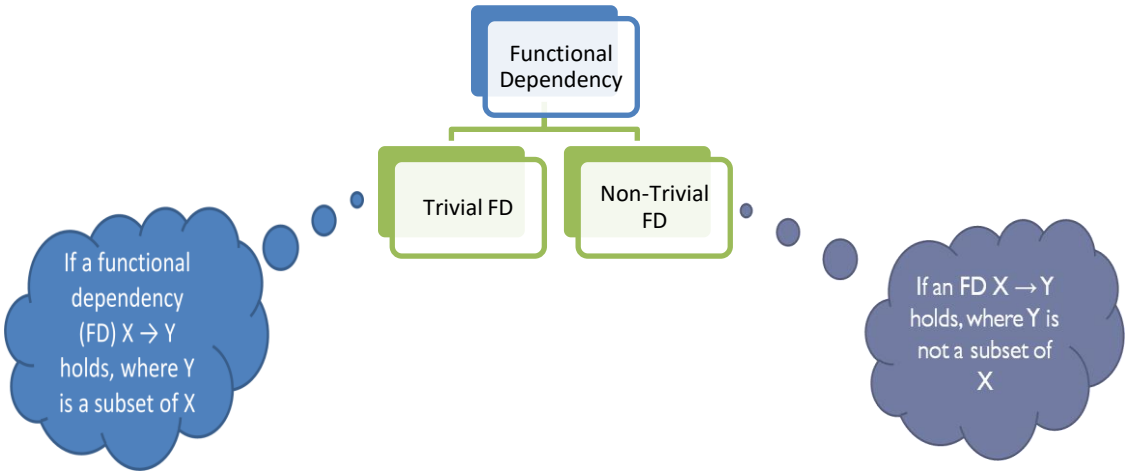
Closure of a set of FDs:

- The set of all FDs implied by a set F of FDs is called closure of F
- denoted as F+
- Three rules, called Armstrong Axioms can be applied repeatedly to infer all FDs implied by a set F of FDs.
- Armstrong Axioms
  - **Reflexivity : if X ⊇ Y, then X → Y**
  - **Augmentation : if X→ Y, then XZ → YZ for any Z**
  - **Transitivity : if X → Y, and Y → Z, then X →Z**
- Some additional rules while reasoning about FDs
  - **Union : if X → Y and X → Z, then X → YZ**
  - **Decomposition : if X → YZ, then X → Y and X → Z**
  - **Pseudo Transitivity Rule : if X → Y and YW → Z, then XW → Z**

Solve these:

- Let F = { A→ B, C → X, BX→Z }
  Show that AC → Z can be inferred from F using inference rules.
  From A→ B and BX→Z, we can apply **Pseudo Transitivity Rule**
  AX → Z ---(1)
  From (1) apply C → X
  AC→Z

- Let F = { A→ B, C → D } and C ⊆ B
  Show that A → D can be inferred from F using inference rules.



Trivial Functional dependency:

The **Trivial dependency** is a set of attributes which are called a trivial if the set of attributes are included in that attribute. So, X -> Y is a trivial functional dependency if Y is a subset of X.

For example:

Emp_id	Emp_name
AS555	Karthiika
AS811	Vedhansh
AS999	Vamsika

Consider this table with two columns Emp_id and Emp_name. {Emp_id, Emp_name} -> Emp_id is a trivial functional dependency as Emp_id is a subset of {Emp_id,Emp_name}.

Non trivial functional dependency:

Functional dependency which also known as a nontrivial dependency occurs when A->B holds true where B is not a subset of A. In a relationship, if attribute B is not a subset of attribute A, then it is considered as a non-trivial dependency.

For example:

Company	CEO	Age
Microsoft	Satya Nadella	51
Google	Sundar Pichai	46
Apple	Tim Cook	38

(Company} -> {CEO} (if we know the Company, we knows the CEO name)
But CEO is not a subset of Company, and hence it's non-trivial functional dependency.

Advantages of Functional Dependency:

- Functional Dependency avoids data redundancy. Therefore same data do not repeat at multiple locations in that database.
- It helps you to maintain the quality of data in the database
- It helps you to defined meanings and constraints of databases
- It helps you to identify bad designs
- It helps you to find the facts regarding the database design

Attribute Closure:

- Attribute Closure of X (X^+) w.r.t to given set of FDs F, *is the set of all attributes functionally determined by X* under the set F.

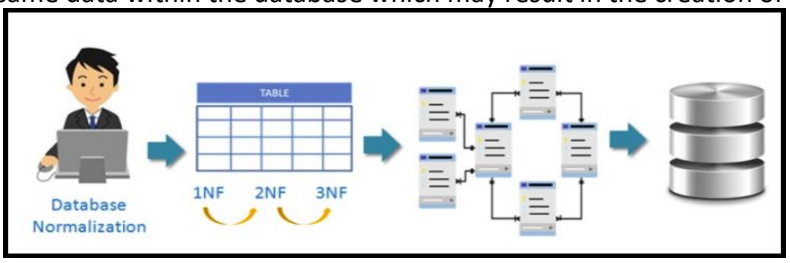
- Algorithm to find out the Attribute closure of attribute set X

closure = *X*;
repeat until there is no change: {
    if there is an FD $U \rightarrow V$ in *F* such that $U \subseteq$ *closure*,
        then set *closure* = *closure* ∪ *V*
}

- Applications of attribute closure.
  - To identify the additional FDs
  - To identify the keys
  - To identify the equivalence of the FDs
  - To identify the minimal set of FDs / irreducible set of FDs / canonical forms of FDs.

What is Normalization?

Normalization is a method of organizing the data in the database which helps you to avoid data redundancy, insertion, update & deletion anomaly. It is a process of analyzing the relation schemas based on their different functional dependencies and primary key. Normalization is inherent to relational database theory. It may have the effect of duplicating the same data within the database which may result in the creation of additional tables.



Normalization of Database :

Database Normalization is a technique of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update and Deletion Anomalies. It is a multistep process that puts data into tabular form, removing duplicated data from the relation tables.

Normalization is used for mainly two purposes,

- Eliminating redundant(useless) data.
- Ensuring data dependencies make sense i.e data is logically stored.

Problems Without Normalization:

If a table is not properly normalized and have data redundancy then it will not only eat up extra memory space but will also make it difficult to handle and update the database, without facing data loss. Insertion, Updation and Deletion Anomalies are very frequent if database is not normalized. To understand these anomalies let us take an example of a Student table.

ROLL NOR	NAME	BRANCH	HOD	PHONE NO
401	ANU	CSE	Mr. X	53337
402	AJAY	CSE	Mr. X	53337
403	RAHUL	CSE	Mr. X	53337
404	KIRAN	CSE	Mr. X	53337

In the table above, we have data of 4 Computer Sci. students. As we can see, data for the fields branch, hod(Head of Department) and office_tel is repeated for the students who are in the same branch in the college, this is **Data Redundancy.**

Insertion Anomaly-

- Suppose for a new admission, until and unless a student opts for a branch, data of the student cannot be inserted, or else we will have to set the branch information as NULL.
- Also, if we have to insert data of 100 students of same branch, then the branch information will be repeated for all those 100 students.
- These scenarios are nothing but Insertion anomalies.

Updation Anomaly - What if Mr. X leaves the college? or is no longer the HOD of computer science department? In that case all the student records will have to be updated, and if by mistake we miss any record, it will lead to data inconsistency. This is Updation anomaly.

Deletion Anomaly- In our Student table, two different informations are kept together, Student information and Branch information. Hence, at the end of the academic year, if student records are deleted, we will also lose the branch information. This is Deletion anomaly.

Candidate Key: Candidate Key is minimal set of attributes of a relation which can be used to identify a tuple uniquely.
Consider student table: ***student (sno, sname, sphone, age)*** we can take **sno** as candidate key.
we can have more than 1 candidate key in a table.
Types of candidate keys:

1. simple (having only one attribute)
2. composite (having multiple attributes as candidate key)

Example: *Finding candidate keys for a table*

R(A, B, C, D)
F = { AB → C, B → D, D → B}
How many candidate keys are there for R?
What are prime attributes in R

$(A)^+ = \{ A \}$
$(AB)^+ = \{ ABCD \}$ ✓
$(AC)^+ = \{ AC \}$
$(AD)^+ = \{ ADBC \}$ ✓

Super keys.....
ABC
ABD
ACD
ABCD

Prime Attributes
A, B, D

Super Key: Super Key is set of attributes of a relation which can be used to identify a tuple uniquely. Adding zero or more attributes to candidate key generates super key.

- A candidate key is a super key but vice versa is not true.
- Consider student table: student(sno, sname,sphone,age) we can take **sno**, **(sno, sname)** as super key.

Prime and non-prime attributes:

✓ A prime attribute is **an attribute that is part of any candidate key**. It can also be used to uniquely identify a tuple in the schema.
✓ A prime attribute in DBMS is also known as a key attribute.

✓ A non-prime attribute is one that is not part of one of the candidate keys.

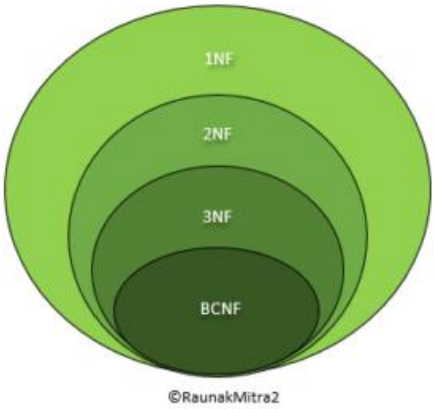
Non Prime Attribute

Roll_No	Name	Age	GPA
1	Arya	21	4
2	Bran	19	3
3	John	24	4.3
4	Max	24	1

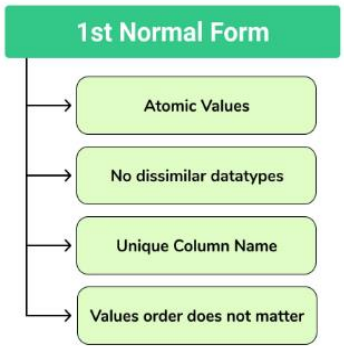
Non Prime Attributes

Normal Forms

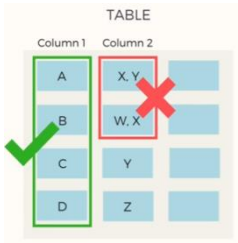
- Provides a guidance to decide whether a database design is good or needs decomposition.
- Can be used to identify the presence of redundancy in relations.
- Normal forms based on FDs are
  - First normal form (1 NF)
  - Second normal form (2 NF)
  - Third normal form (3 NF)
  - Boyce-Codd normal form (BCNF)
- A relational table is said to be in a particular normal form if it satisfied a certain set of constraints.
- If a relation is in one the of the higher normal forms, then it will be automatically in all lower normal forms
- The higher the normal form the lower the redundancy. Therefore higher normal forms are preferable.



First Normal Form (1 NF):

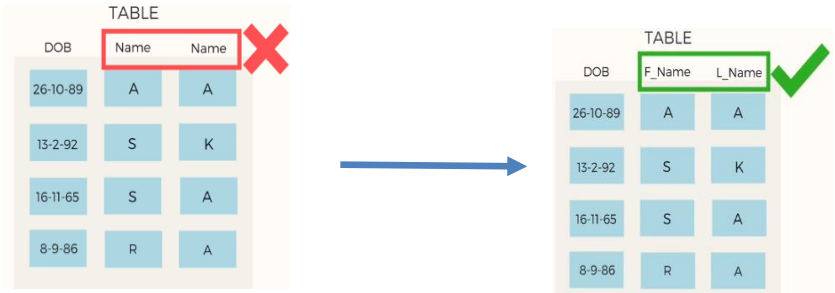


It is a level of normalization in DBMS. A relation is said to be in 1 normal form in DBMS (or 1NF) when it consists of an atomic value. In simpler words, 1NF states that a table's attribute would not be able to hold various values- it will only be able to hold an attribute of a single value.

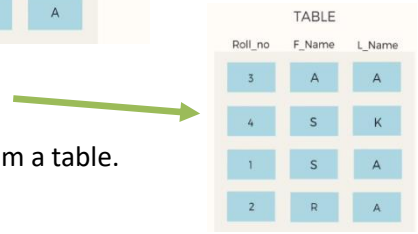


➢ A schema is in First normal form
  - There are 4 basic rules that a table should follow to be in 1st Normal Form
  - **Rule 1 :**
    - if the domain of every attribute is atomic
      - That is, no composite values (lists or sets)
      - Entries like X,Y and W,X violates the rule

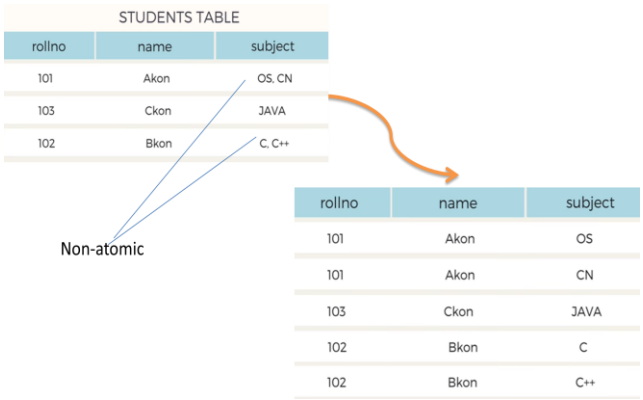
  - **Rule 2:**
    - All entries in any column must be of the same kind.
    - Do not inter-mix different types of values in any column
  - **Rule 3:**
    - Each column must have a unique name
    - Same name leads to confusion at the time of data retrieval



- Rule 4:
  - Order in which data stored doesn't matter.
  - No two rows are identical
  - Using SQL query, we can easily fetch data in any order from a table.



Example:



This way, although a few values are getting repeated, we can still see that there is just one value in every column.

Problem Table:

- Consider a table where a student can have multiple phone numbers

Roll no	Name	Phone
66	Trishaank	P1
73	Prashant	P2, P3
79	Sanjay	P4
82	Srinivas	P5

Atomic values

Divide the table into two parts such that all the ***multivalued attributes in one table at single-valued attributes in another table*** and add primary key attribute of the original table to each newly formed table

Roll no	Phone
66	P1
73	P2
73	P3
79	P4
82	P5

Roll no	Name
66	Trishaank
73	Prashant
79	Sanjay
82	Srinivas

Partial Dependency:

A functional dependency X->Y is a partial dependency if Y is functionally dependent on X and Y can be determined by any proper subset of X.

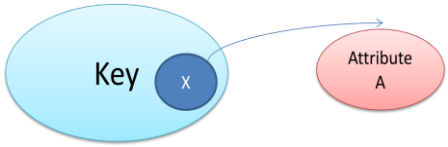
For example, we have a relationship AC->B, A->D, and D->B.

Now if we compute the closure of $\{A^+\}$ = ADB

Here A is alone capable of determining B, which means B is partially dependent on AC.

Example:

A is not part of key X-> A is a partial dependency



Let us take another example –

name	roll_no	course
Ravi	2	DBMS
Tim	3	OS
John	5	Java

Here, we can see that both the attributes name and roll_no alone are able to uniquely identify a course. Hence we can say that the relationship is partially dependent.

Full Functional Dependency:

In X → Y,

if Y cannot be determined by any of subsets of X, then Y is said to be fully functionally dependent on X

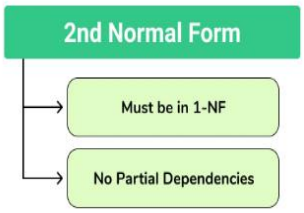
Example – In the relation ABC->D, attribute D is fully functionally dependent on ABC and not on any proper subset of ABC. That means that subsets of ABC like AB, BC, A, B, etc cannot determine D.

supplier_id	item_id	price
1	1	540
2	1	545
1	2	200
2	2	201

From the table, we can clearly see that neither supplier_id nor item_id can uniquely determine the price but both supplier_id and item_id together can do so. So we can say that price is fully functionally dependent on { supplier_id, item_id }. This summarizes and gives our fully functional dependency – { supplier_id , item_id } -> price.

Full Functional Dependency	Partial Functional Dependency
A functional dependency X->Y is a fully functional dependency if Y is functionally dependent on X and Y is not functionally dependent on any proper subset of X.	A functional dependency X->Y is a partial dependency if Y is functionally dependent on X and Y can be determined by any proper subset of X.
In full functional dependency, the non-prime attribute is functionally dependent on the candidate key.	In partial functional dependency, the non-prime attribute is functionally dependent on part of a candidate key.
In fully functional dependency, if we remove any attribute of X, then the dependency will not exist anymore.	In partial functional dependency, if we remove any attribute of X, then the dependency will still exist.
Full Functional Dependency equates to the normalization standard of Second Normal Form.	Partial Functional Dependency does not equate to the normalization standard of Second Normal Form. Rather, 2NF eliminates the Partial Dependency.
An attribute A is fully functional dependent on another attribute B if it is functionally dependent on that attribute, and not on any part (subset) of it.	An attribute A is partially functional dependent on other attribute B if it is functionally dependent on any part (subset) of that attribute.
Functional dependency enhances the quality of the data in our database.	Partial dependency does not enhance the data quality. It must be eliminated in order to normalize in the second normal form.

Second Normal Form:



It is a normalization level in DBMS. A relation is said to be in the 2nd Normal Form in DBMS (or 2NF) when it is in the First Normal Form but has no non-prime attribute functionally dependent on any candidate key's proper subset in a relation. A relation's non-prime attribute refers to that attribute that isn't a part of a relation's candidate key.

Uses of Second Normal Form in DBMS:

The concept of 2nd Normal Form in DBMS depends on full functional dependency. We apply 2NF on the relations that have composite keys or the relations that have a primary key consisting of two attributes or more. Thus, the relations having a primary key of a single attribute automatically get to their 2NF. Any relation that doesn't exist in the 2NF may eventually suffer from further update anomalies.

Not allowed FDs –

```
prime -> non prime
```

If say your functional dependency is of the form A->X where 'A' is a prime attribute but not a key and 'X' is a non prime attribute, then such an FD is not allowed in 2NF.

Allowed FDs –

```
Prime -> Prime
Non prime -> Prime/Non prime
Key -> Prime/Non prime
Prime + Non prime combination -> Prime/Non Prime
```

1. ***Is this relation in 2 NF***

   Consider the relational schema R (A, B, C, D) and the functional dependencies.

   F = { AB → C, BC → D, A → D}

   - *It is in 1NF*
   - *2NF:*

   $(A)^+ = \{A,D\}$
   $(B)^+ = \{B\}$
   $(C)^+ = \{C\}$
   $(D)^+ = \{D\}$

   $(AB)^+ = \{A,B,C,D\}$ ✓
   $(AC)^+ = \{A,C,D\}$
   $(AD)^+ = \{A,D\}$
   $(BC)^+ = \{B,C,D\}$
   $(BD)^+ = \{B,D\}$
   $(CD)^+ = \{C,D\}$

   Prime Attributes — A, B

   Non-Prime Attributes — C, D

   Clearly, prime attributes for Relation R are: {A,B} while non-prime attributes are: {C,D}.

   Check the partial Dependencies:

AB → C	✓	*AB is a key*
BC → D	✓	*A composite key determines non-prime attribute*
A → D	✗	*Partial dependency*

   As the partial dependency exists, therefore it is not in 2NF.

2. Consider schema R = (A, B, C, G, H, I) and the set F of functional dependencies {A→ B, A → C, CG → H, CG → I, B → H}. Compute the candidate keys of the schema. Check whether the schema is in 2NF or not

i. It is in 1NF

ii. 2NF:

$(A)^+ = \{A,B,C,H\}$
$(B)^+ = \{B,H\}$
$(C)^+ = \{C\}$
$(G)^+ = \{G\}$
$(H)^+ = \{H\}$
$(I)^+ = \{I\}$
$(AB)^+ = \{A,B,H\}$
$(AC)^+ = \{A,C,B,H\}$
$(AG)^+ = \{A,G,B,C,H,I\}$ ✓
$(AH)^+ = \{A,H,B,C\}$
$(HI)^+ = \{H,I\}$
$(AI)^+ = \{A,I,B,C,H\}$
$(BC)^+ = \{B,C,H\}$
$(BG)^+ = \{B,G,H\}$
$(BH)^+ = \{B,H\}$
$(BI)^+ = \{B,I,H\}$
$(CG)^+ = \{C,G,H,I\}$
$(CH)^+ = \{C,H\}$
$(CI)^+ = \{C,I\}$
$(GH)^+ = \{G,H\}$
$(GI)^+ = \{G,I\}$

Prime Attributes — A, G

Non-Prime Attributes — B, C, H, I

Clearly, prime attributes for Relation R are: {A,G} while non-prime attributes are: {B,C,H,I}.

Check the partial Dependencies:

$A \rightarrow B$	✗	*Partial dependency*
$A \rightarrow C$	✗	*Partial dependency*
$CG \rightarrow H$	✓	*A composite key determines non-prime attribute*
$CG \rightarrow I$	✓	*A composite key determines non-prime attribute*
$B \rightarrow H$	✓	*non-prime attribute determines prime attribute*

As the partial dependency exists, therefore it is not in 2NF.

3. Decompose the following table into 2NF:

Problem: Score Table

Student_id	Subject_id	Marks	teacher
191	13A	70	Java Teacher
191	15D	78	DBMS teacher
181	13A	80	Java Teacher

- So even if the table is in First Normal Form it suffers from problems due to insertion, updation and deletion hence we need to go for second normal form.
- Dependency are
  - Student_id, subject_id → marks
  - Subject_id → teacher
- Identify the partial and full dependencies and apply decomposition rule.

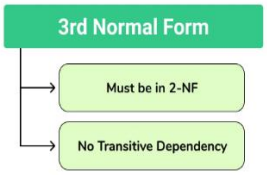
In the above table column teacher is dependent on subject_id

- The simple solution is to create a separate table for subject_id and teacher and removing it from Score table

Subject_id	teacher
13A	Java Teacher
15D	DBMS teacher
13A	Java Teacher

Student_id	Subject_id	Marks
191	13A	70
191	15D	78
181	13A	80

Third Normal Form:



A relation (table) is said to in 3NF:

- If it is in 2NF.
- There must be no transitive functional dependency for non prime attributes.

Transitive FD : "If non prime attribute is dependent on another non prime attribute then it is called transitive functional dependency."

- Let R be a relation schema,
- F be the set of FDs given to hold over R,
- X be a subset of the attributes of R,
- And A be an attribute of R.
- R is in third normal form if, for every FD X → A in F, one of the following statements is true:
- that is, it is a trivial FD, or
  - X is a super key , or
  - A is part of some key for R (prime attribute)

In simple words
- R should be in 2 NF
- No NPA should be transitively dependent on Candidate key

Q.1 Check whether the following relations are in 3NF or not R (A, B, C)

F = { A → B, B → C}

Solution: Firstly find the candidate key in the relation:

$(A)^+ = ABC$

A is the candidate key, because closure of A has all the attributes of R.

Prime Attributes *Non-Prime Attributes*

Clearly, prime attributes for Relation R are: {A} while non-prime attributes are: {B, C}.

A relation is said to be **2NF**, if it is 1NF and for dependency X -> a, there should not be any partial dependency.

Check the partial Dependencies:

A → B ✓ *fully functional dependency*

B → C ✓ *Non-prime determines non-prime attribute*

As the no partial dependency exists , therefore it is in 2NF.

A relation is said to be **3NF**, if it holds at least one of the following for every non trivial functional dependency X -> a:

X is super key.

a is prime attribute.

	Trivial Dependency		Super Key	Prime Attributes
A -> B	No	→	A ✓	B ✗
B -> C	No	→	B ✗	C ✗

A -> B – A is super key and B is not a prime attribute.
B -> C -- Neither B is super key, nor C is prime attribute.
So, the relation is not in 3NF as it is not following the rules of 3NF.

Therefore, R(ABC) needs to be divided into following:
R1(A B)
R2 (B C)
Now R1 and R2 are in 3NF.

Q.2 Suppose a relational schema R (A B C D E) and set of functional dependencies
F: { A -> B B-> E C-> D }
Check out that relation is in 3NF or not? If not decompose it in 3NF.

Solution: Firstly find the candidate key in the relation:
$(AC)^+$ = ABCDE
AC is the candidate key, because closure of AC has all the attributes of R.

Prime Attributes *Non-Prime Attributes*

Clearly, prime attributes for Relation R are: {A,C} while non-prime attributes are: {B,D,E}.

A relation is said to be 3NF, if it holds at least one of the following for every non trivial functional dependency X -> a:
X is super key.
a is prime attribute.
A -> B – Neither A is super key nor B is prime attribute.
B -> E -- Neither B is super key, nor E is prime attribute.
C-> D -- Neither C is super key, nor D is prime attribute.

	Trivial Dependency		Super Key	Prime Attributes
A -> B	No	→	A ✗	B ✗
B -> E	No	→	B ✗	E ✗
C -> D	No	→	C ✗	D ✗

So, the relation is not in 3NF as it is not following the rules of 3NF.
Therefore, R(ABCDE) needs to be divided into following:
R1(A B E) → R11 (A B)
R12 (B E)
R2 (C D)
R3(A C)
Now R11, R12, R2, R3 are in 3NF.

Surrogate Key

- A artificial primary key which is generated automatically by the system
- The value of surrogate key is numeric
- It is automatically incremented for each new row
- Surrogate key is called the factless key as it is added just for our ease of identification of unique values and contains no relevant fact (or information) that is useful for the table.
- Example: Suppose we have two tables of two different colleges having the same column registration_no , name and percentage , each table having its own natural primary key, that is registration_no.

Table of College VIT

registration_no	name	percentage
210101	Harry	90
210102	Maxwell	65
210103	Lee	87
210104	Chris	76

Table of College SVECW

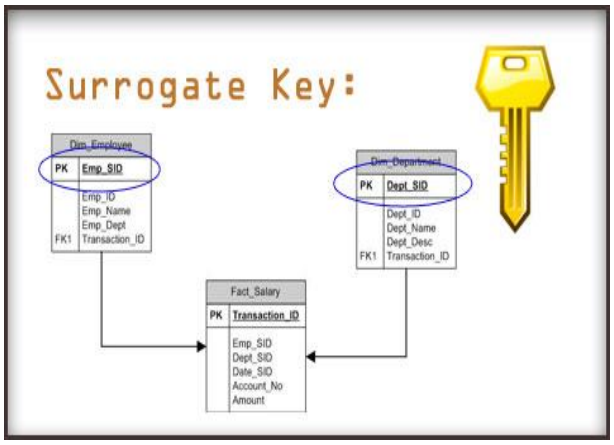
registration_no	name	percentage
CS107	Taylor	49
CS108	Simon	86
CS109	Sam	96
CS110	Andy	58

- Now, suppose we want to merge the details of both
- Resulting table will be –

surr_no	registration_no	name	percentage
1	210101	Harry	90
2	210102	Maxwell	65
3	210103	Lee	87
4	210104	Chris	76
5	CS107	Taylor	49
6	CS108	Simon	86
7	CS109	Sam	96
8	CS110	Andy	58

Registration_no cannot be the primary key of the table as it does not match with all the records of the table though it is holding all unique values of the table .

Now , in this case, we have to artificially primary key for this table. We can do this by adding a column surr_no in the table that contains anonymous integers and has no direct relation with other columns .



Some examples of Surrogate key are :

- System date & time stamp
- Random alphanumeric string

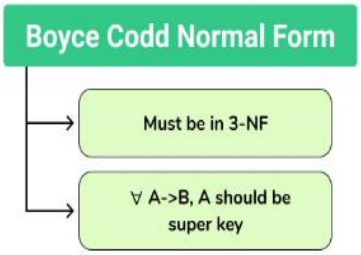
Advantages of the surrogate key :

- Enables us to run fast queries
- Performance is enhanced as the value of the key is relatively smaller.
- The key value is guaranteed to contain unique information .

Disadvantages of the surrogate key :

- Can never be used as a search key.
- As the key value has no relation to the data of the table, so third normal form is violated.
- The extra column for surrogate key will require extra disk space.
- We will need extra IO when we have to insert or update data of the table.

Boyce-Codd Normal Form:



- Let R be the relation schema, F be the set of FDs that hold on R we say that R is in BCNF w.r.t F, iff, for every FD in F, one of the following 2 rules conditions is true.
  - A ϵ X if X →A is a trivial FD or
  - X is a super key.
- BCNF is based on functional dependencies, and all the candidate keys of the relation are taken into consideration. BCNF is stricter than 3NF and has some additional constraints along with the general definition of 3NF.

Example: Consider a relation R with attributes (student, subject, teacher).

Student	Teacher	Subject
Jhansi	P.Naresh	Database
jhansi	K.Das	C
subbu	P.Naresh	Database
subbu	R.Prasad	C

F: { **(student, Teacher) -> subject**
(student, subject) -> Teacher
Teacher -> subject}

- Candidate keys are (student, teacher) and (student, subject).
- The above relation is in 3NF [since there is no transitive dependency]. A relation R is in BCNF if for every non-trivial FD X->Y, X must be a key.
- The above relation is not in BCNF, because in the FD (teacher->subject), teacher is not a key. This relation suffers with anomalies –
- For example, if we try to delete the student Subbu, we will lose the information that R. Prasad teaches C. These difficulties are caused by the fact the teacher is determinant but not a candidate key.

Decomposition for BCNF

Teacher-> subject violates BCNF [since teacher is not a candidate key].

If X->Y violates BCNF then divide R into R1(X, Y) and R2(R-Y).

So R is divided into two relations R1(Teacher, subject) and R2(student, Teacher).

R1

Teacher	Subject
P.Naresh	database
K.DAS	C
R.Prasad	C

R2

Student	Teacher
Jhansi	P.Naresh
Jhansi	K.Das
Subbu	P.Naresh
Subbu	R.Prasad

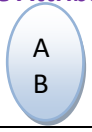
All the anomalies which were present in R, now removed in the above two relations.

Example: find the highest normalization form, and for that, we are given a relation **R(A, B, C)** with functional dependencies as follows: {AB ->C, C ->B, AB ->B}

Sol: $(AB)^+ = \{A,B,C\}$

Candidate Key (given): **{AB}**

Prime Attributes



Non-Prime Attributes

Clearly, prime attributes for Relation R are: {A,B} while non-prime attributes are: {C}.

- **It is in First Normal Form**
- **Second normal form**

	Partial Dependency
AB->C	No ✗ (fully functionally dependencies)
AB->B	No ✗ (fully functionally dependencies)
C->B	No ✗ (non-prime attribute determines prime attribute)

As there is no partial dependency , the given relation schema is in **2NF.**

- **Third normal form**

	Trivial Dependency	Super Key	Prime attribute
AB->C	No ✗	AB ✓	C ✗
AB->B	Yes ✓	AB ✓	B ✓
C->B	No ✗	C ✗	B ✓

As there is no transitive dependency , the given relation schema is in **3NF.**

- **Boyce-Codd normal form**

	Trivial Dependency	Super Key
AB->C	No ✗	AB ✓
AB->B	Yes ✓	AB ✓
C->B	No ✗	C ✗

- Clearly, {AB->C} and {AB->B} are in BCNF because
  - AB is the candidate key present on the LHS of both dependencies.
  - The second dependency, {C->B}, however, is not in BCNF because C is neither a super key nor a candidate key.
- C->B is, however, present in 3NF because B is a prime attribute that satisfies the conditions of 3NF. Hence, relation R has 3NF as the highest normalization form.

Decomposition:

R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R. Intuitively, we want to store the information in any given instance of R by storing projections of the instance. This section examines the use of decompositions.

- A tool that allows us to eliminate redundancy.
- Properties of decomposition
  - Lossless-Join Decomposition
  - Dependency-Preserving Decomposition

- **Lossless-Join Decomposition:** Let R be a relation schema, F be the set of FDs that hold on R.

- The decomposition of R into 2 sub-schemas with attribute sets X and Y is said to be loss-less join decomposition w.r.t. 'F', iff, for every instance 'r' of 'R' that satisfies the dependencies in F, the following condition should be true.

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- i.e., we should be able to recover the original relation from decomposed relations
- Loss-less decomposition also called as non-additive join decomposition.
- No extraneous tuples appear after joining of the sub-relations.

- Decomposition of $r = (A, B, C)$ into:
  $r_1 = (A, B)$ and $r_2 = (B, C)$

A	B	C
α	1	A
β	2	B

r

A	B
α	1
β	2

$\Pi_{A,B}(r)$

B	C
1	A
2	B

$\Pi_{B,C}(r)$

$\Pi_{r_1}(r) \bowtie \Pi_{r_2}(r)$

A	B	C
α	1	A
β	2	B

This decomposition is loss-less join decomposition

As it retains the original relation after joining the decomposed relations.

Note: All decompositions used to eliminate redundancy must be lossless join decomposition.

Let R be a relation and F be a set of FDs that hold over R. decomposition of R into relations with attribute sets R_1 and R_2 is lossless if and only if F^+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ or the FD $R_1 \cap R_2 \rightarrow R_2$

In other words
R_1 and R_2 must have one attribute in common and
The common attribute must be a super key for either R_1 or R_2

Example 1:

R(A, B, C, D)
F = { A →B, B → C, C→A}
R is decomposed into R1(A,B,C) and R2(C,D).
Find whether the decomposition is loss-less or lossy join decomposition?

Solution:

- **Attributes(R1) ∪ Attributes(R2) = Attributes(R)**
  - {A, B, C} ∪ {C,D} = {A,B,C,D} = Attributes(R) ✓
- **Attributes(R1) ∩ Attributes(R2) ≠ ϕ**
  - {A, B, C} ∩ {C,D} = {C} ≠ ϕ ✓
- **Attributes(R1) ∩ Attributes(R2) = Super key of either R1 or R2**

  {A, B, C} ∩ {C, D} = {C}
  R1(A,B,C) :
  (C)$^+$ = {C, A, B} can determine all the attributes of R1, C is a super key for R1 ✓
  R2(C,D) :
  (C)$^+$ = {C, A, B} cannot determine all the attributes of R2, C is not a super key for R2 ✗

All the properties were satisified. Therefore it is a loss-less join decomposition.

Example 2:

R(X, Y, Z)
F = { X →Y, Y → Z}
R is decomposed into R1(X,Y) and R2(Y,Z). Find whether the decomposition is loss-less or lossy?

Solution:

- **Attributes(R1) ∪ Attributes(R2) = Attributes(R)**
  - {X,Y} ∪ {Y,Z} = {X,Y,Z} = Attributes(R) ✓
- **Attributes(R1) ∩ Attributes(R2) ≠ ϕ**
  - {X,Y} ∩ {Y,Z} = {Y} ≠ ϕ ✓
- **Attributes(R1) ∩ Attributes(R2) = Super key of either R1 or R2**
  - {X,Y} ∩ {Y,Z} = {Y}
    R1(X,Y) :
    $(Y)^+$ = {Y} cannot determine X, Y soo it is not a super key for R1 ✗
    R2(Y,Z)
    $(Y)^+$ = {Y,Z} can determine all the attributes of R2, Y is a super key for R2 ✓

All the properties were satisified. Therefore it is a loss-less join decomposition.

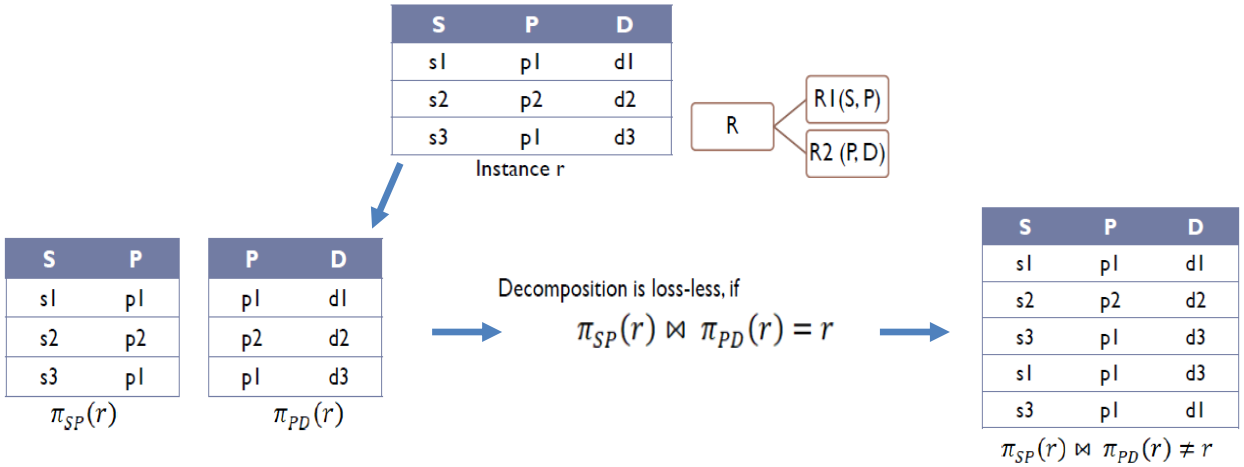
Lossy- Join decomposition: A relation X is decomposed into sub relations X_1, X_2, … X_n. This decomposition is called lossy join decomposition. When the join of the sub relations does not result in the same relation R that was decomposed.

- Contains extraneous tuples

$$X1 \bowtie X2 \bowtie X3 \ldots\ldots \bowtie Xn \supset X$$

Here, the operator ⋈ acts as a natural join operator.

- Lossy decomposition also called as "careless decomposition".
- Because of extraneous tuples, identification of the original tuples is difficult.
- Example:



This decomposition is not loss-less
It is a lossy decomposition.

Dependency Preserving Decomposition:

- Let R be a relation schema and F be the set of FDs that hold over R.
- The decomposition of R into sub schemas with attribute sets X and Y is dependency preserving, if

$$(F_X \cup F_Y)^+ = F^+$$

  - F_X is the set of FDs in F^+ that includes only attributes of X
  - F_Y is the set of FDs in F^+ that includes only attributes of Y

Intuitively, a dependency-preserving decomposition allows us to enforce all FDs by examining a single relation instance on each insertion or modification of a tuple.

- In the dependency preservation, **at least one decomposed table must satisfy every dependency**. If a relation R is decomposed into relation R1 and R2, then the dependencies of R either must be a part of R1 or R2 or must be derivable from the combination of functional dependencies of R1 and R2.

Example 1:

R (A, B, C)

F = { A → B, B → C, C→A} R is decomposed into R1(A,B) and R2(B,C). Find whether the decomposition is dependency preserving or not?

Solution: The decomposition of R into sub schemas with attribute sets X and Y is dependency preserving,if

$$(F_{R1} \cup F_{R2})^+ = F^+$$

R1(A,B)			R2(B,C)		
$(A)^+ = \{A,B,C\}$	$(A)^+ = \{A,B,C\}$ (skip the attributes which are not in the R1 relation and also same attribute of left side).	**A → B**	$(B)^+ = \{B,C,A\}$	$(B)^+ = \{B,C,A\}$ (skip the attributes which are not in the R2 relation and also same attribute of left side).	**B → C**
$(B)^+ = \{B,C,A\}$	$(B)^+ = \{B,C,A\}$ (skip the attributes which are not in the R1 relation and also same attribute of left side).	**B → A**	$(C)^+ = \{C,A,B\}$	$(C)^+ = \{C,A,B\}$ (skip the attributes which are not in the R2 relation and also same attribute of left side).	**C → B**
$(AB)^+ = \{A,B,C\}$	A is determining all the attributes of R1, so no need to check the combinations	-	$(BC)^+ = \{B,C,A\}$	B is determining all the attributes of R2, so no need to check the combinations	-

$(F_{R1} \cup F_{R2})$ = { **A → B, B → A, B → C, C → B** }

Now we have to check whether $(F_{R1} \cup F_{R2})$ covers F

Consider functional dependencies of R

F : FD's of R	$(F_{R1} \cup F_{R2})$ covers F { A → B, B → A, B → C, C → B } Covers F
A → B	$(A)^+ = \{A,B,C\}$ ✓
B → C	$(B)^+ = \{B,C,A\}$ ✓
C→ A	$(C)^+ = \{A,B,C\}$ ✓

- As all the original dependencies are available in decomposed dependencies. Therefore we can say that it is a dependency preserving decomposition.

Multi-value dependency

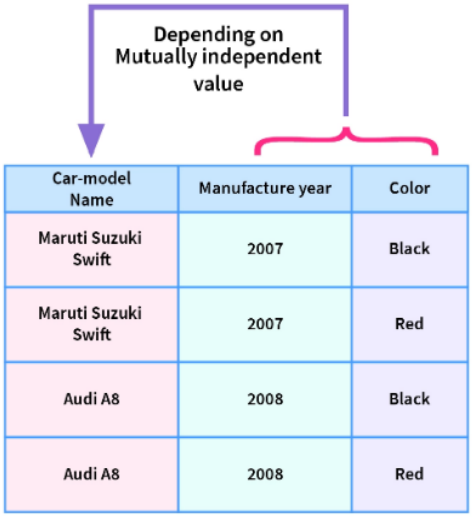
- Multivalued dependency would occur whenever two separate attributes in a given table happen to be independent of each other. And yet, both of these depend on another third attribute.
- The multivalued dependency contains at least two of the attributes dependent on the third attribute. This is the reason why it always consists of at least three of the attributes.

Conditions for Multivalued Dependency in DBMS

- An MVD would mean that for some single value of the attribute 'x', multiple values of attribute 'y' can exist. Thus, we will write it as follows:

  x –> –> y
- So, it is actually read as: x is multivalued dependent only.

Example: Suppose that there is a car manufacturing company that produces two of the colors in the market, i.e., red and black of each of their models.



Car-model Name	Manufacture year	Color
Maruti Suzuki Swift	2007	Black
Maruti Suzuki Swift	2007	Red
Audi A8	2008	Black
Audi A8	2008	Red

In this case, the columns COLOR and MANUFACTURE YEAR are dependent on CAR_MODEL NAME, and they are independent of each other. Thus, we can call both of these columns multivalued. These are, as a result, dependent on CAR_MODEL NAME. Here is a representation of the dependencies we discussed above:

CAR_MODEL NAME →→ MANUFACTURE YEAR

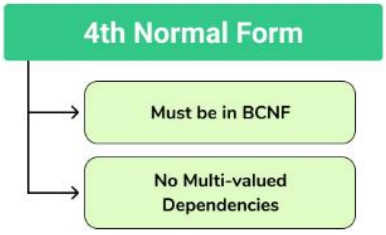
CAR_MODEL NAME →→ COLOR

We can read this as "CAR_MODEL NAME multidetermined MANUFACTURE YEAR" and "CAR_MODEL NAME multidetermined COLOUR".

We always use multi-valued conditions when we encounter these two different ways:

- When we want to test the relations or decide if these happen to be lawful under some arrangement of practical as well as multi-valued dependencies.
- When we want to determine what limitations are there on the arrangement of the lawful relations. Thus, we will concern ourselves with just the relations that fulfil a given arrangement of practical as well as multi-valued dependencies.

FOURTH NORMAL FORM:



- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.
- For a dependency A → B, if for a single value of A, multiple values of B exists, then the relation will be a multi-valued dependency.

Examples of 4NF:

STU_ID	COURSE	HOBBY
21	Computer	Dancing
21	Math	Singing
34	Chemistry	Dancing
74	Biology	Cricket
59	Physics	Hockey

The given STUDENT table is in 3NF, but the COURSE and HOBBY are two independent entity. Hence, there is no relationship between COURSE and HOBBY.

In the STUDENT relation, a student with STU_ID, **21** contains two courses, **Computer** and **Math** and two hobbies, **Dancing** and **Singing**. So there is a Multi-valued dependency on STU_ID, which leads to unnecessary repetition of data. So to make the above table into 4NF, we can decompose it into two tables:

STUDENT_HOBBY

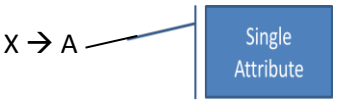
STU_ID	HOBBY
21	Dancing
21	Singing
34	Dancing
74	Cricket
59	Hockey

STUDENT_COURSE

STU_ID	COURSE
21	Computer
21	Math
34	Chemistry
74	Biology
59	Physics

Decomposition into BCNF

- Let R be the relation schema and F be the set of FDs that hold on R.
- Assume that R is not in BCNF



  - Replace R with (R-A), XA
  - Apply this algorithm recursively.

Example: Let R(A,B,C,D,E) be a relational schema with set of FD as
F : { A -> BC , CD -> E, B->D, E -> A} decompose into bcnf
Sol:

A -> BC ---- A -> B and A -> C

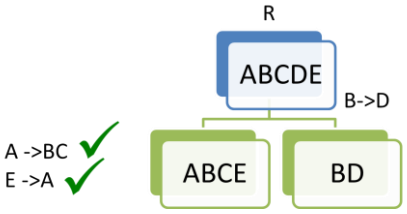
LHS should be a super key

$(A)^+ = \{A,B,C,D,E\}$ A -> BC ✓

$(CD)^+ = \{C,D,E,A,B\}$ CD -> E ✓

$(B)^+ = \{B,D\}$ B->D ✗

$(E)^+ = \{E,A,B,C,D\}$ E -> A ✓



Decomposition into 3NF:

1. Dependent reserving decomposition into 3NF.
2. 3NF Synthesis

Algorithm:

Step 1: Eliminate redundant FDs, resulting in a canonical cover F_c of F.
Step 2: Create a relation $R_i = XY$ for each FD $X \rightarrow Y$ in F_c
Step 3: If the key of R does not occur in any relation R_i, create one more relation $R_i = K$.

Example: Let R(A,B,C,D,E) be a relational schema with FDs as
F = { A → B , BC → D, A → C } convert into 3NF.

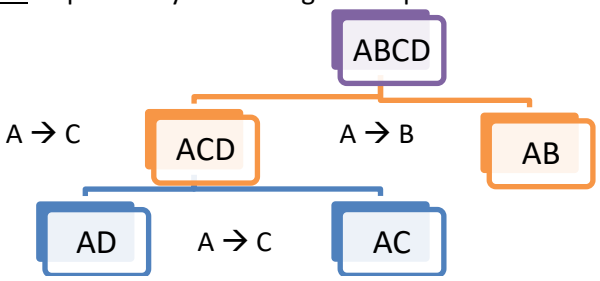
Solution: Candidate key – AE

Checking for 3NF:

A → B (violates 3NF rules)
Therefore R is not in 3NF.

3NF decomposition:

Method 1: Dependency Preserving decomposition into 3NF.



Therefore Loss-less join decomposition of R is R1(A,D,E) R2(A,C) R3(A,B)

$(F_{ADE} \cup F_{AC} \cup F_{AB})^+ = F^+$

A → B , BC → D, A → C

R1(A,D,E)		R2(A,C)		R3(A,B)	
$(A)^+ = \{A,B,C,D\}$	**A -> D**	$(A)^+ = \{A,B,C,D\}$	**A → C**	$(A)^+ = \{A,B,C,D\}$	**A → B**
$(D)^+ = \{D\}$	-	$(C)^+ = \{C\}$	-	$(B)^+ = \{B\}$	-
$(E)^+ = \{E\}$	-				
$(AD)^+ = \{A,D,B,C\}$	-				
$(AE)^+ = \{A,E,B,C,D\}$	**AE -> D**				

$(F_{ADE} \cup F_{AC} \cup F_{AB})^+ = \{A\text{->}D, AE \text{ -> } D, A \text{ -> } C, A\text{-> } B\}$

The functional dependencies of Relational schema are F =

A → B ✓

BC → D $(BC)^+ = \{B,C\}$ ✗

A → C ✓

$(F_{ADE} \cup F_{AC} \cup F_{AB})^+$ not equal to F^+

Here decomposition of R into R1,R2 and R3 is not dependency preserving and BC -> D not enforcing, so we add another table R4(B,C,D)

The decomposed relations are

R1(A,D,E)
R2(A,C)
R3(A,B)
R4(B,C,D)

} Loss-less and dependency preserving

Method 2: 3NF Synthesis

F = { A → B , BC → D, A → C }

The decomposed relations can be R1(A,B) R2(B,C,D) R3 (A,C)

R1(A,B)		R2(B,C,D)		R3(A,C)	
$(A)+ = \{A,B,C,D\}$	**A → B**	$(B)^+ = \{B\}$		$(A)^+ = \{A,B,C,D\}$	**A → C**
$(B)+ = \{B\}$	-	$(C)^+ = \{C\}$		$(C)^+ = \{C\}$	
		$(D)^+ = \{D\}$			
		$(BC)^+ = \{B,C,D\}$	**BC->D**		
		$(CD)^+ = \{C,D\}$			

$(F_{AB} \cup F_{BCD} \cup F_{AC})^+ = \{A\text{->}B, BC \text{ -> } D, A \text{ -> } C\}$

The functional dependencies of Relational schema are F =

A → B ✓

BC → D

A → C ✓

$(F_{ADE} \cup F_{AC} \cup F_{AB})^+ = F^+$

Here decomposition of R into R1,R2 and R3 is dependency preserving.

Now determine loss-less or not

	A	B	C	D	E
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}

A -> B

	A	B	C	D	E
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	a_1	a_2	a_3	b_{34}	b_{35}

A -> C

	A	B	C	D	E
R_1	a_1	a_2	a_3	b_{14}	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	a_1	a_2	a_3	b_{34}	b_{35}

BC -> D

	A	B	C	D	E
R_1	a_1	a_2	a_3	a_4	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	a_1	a_2	a_3	a_4	b_{35}

No single row contains all 'a' values. So the decomposition is not loss-less. In order to make it loss-less, add any key of original table R_4(A,E).

The decomposed relations are

R1(A,B)
R2(B,C,D)
R3(A,C)
R4(A,E)

} Loss-less and dependency preserving

QUESTION BANK

1. What is Normalization ? Explain 1NF with Example.
2. Explain different anomalies handled using Normalization with suitable examples.
3. Consider the Table:
   EMP(EMP_ID,ENAME,SALARY,AADHAR,MOBILE,HOBBIES)
   Normalize the table to 2NF
4. What is functional Dependency? Explain its properties along with a example.
5. Why is normalization needed? Explain the process of normalization.
6. What is partial dependency? Explain how to use normalization to avoid it.
7. Consider schema R = (A, B, C, G, H, I) and the set F of functional dependencies {A→ B, A → C, CG → H, CG → I, B → H}. Compute the candidate keys of the schema. Check whether the schema is in 2NF or not
8. Consider the table : Table_Name : STUDENT

Stud_ID	Stud_Name	Aadhar	Section	Hobbies
1	Akshar	499977268949	A	Reading Books, Swimming
2	Nayan	315165254889	B	Music, Fiction Stories
3	Ritu	133126294520	B	Swimming

4	Kumar	488877295645	C	Yoga

Perform Normalization on the above table so that it can be in 1NF

9. Consider the table : Table_Name : PRODUCT

Product_Id	Product_Name	Price	Stock_Available
101	Apple	65000,70000	3
102	Mac	50100	3
104	Dell	45000,24000	3
103	HP	32000	3

Perform normalization on the above table and bring it into 2NF.

10. What is meant by the closure of functional dependencies? Illustrate with an example.
11. Explain briefly about 3NF, 4NF and BCNF with suitable examples?
12. Explain about Boyce Codd normal form with an example.
13. What is Functional Dependency? Explain types and properties of FD's.
14. Given a Relation R=(X,Y,Z) and Functional Dependencies are
    F={ {X,Y}→{Z}, {Z}→{X} }
    Determine all Candidate keys of R and the normal form of R with proper explanation.
15. How to compute closure of set of functional dependency? Explain with a suitable example schema.
16. What is multi valued dependency? State and explain fourth normal form based on this concept.
17. Explain the advantages of decomposition? Discuss the problems faced in decomposition.
18. Explain the role of functional dependencies in normalization with suitable examples.
19. State BCNF. How does it differ from 3NF?
20. Define functional dependency? How can you compute the minimal cover for a set of functional dependencies? Explain it with an example.
21. Elaborate the importance of computing closure of functional dependencies. Explain the procedure with an example.
22. What is lossless join decomposition? Explain the same with an example.
23. Define normalization. Explain the conditions that are required for a relation to be in 2NF, 3NF and BCNF with suitable examples.
24. What is lossless join decomposition? Explain the same with an example.
25. Explain the problems related to decomposition.
26. Explain BCNF and the properties of decompositions.
27. Explain FOURTH and THIRD normal forms with examples.
28. Elaborate the importance of computing closure of functional dependencies. Explain the procedure with an example.
29. Given Relation, R=(A,B,C,D,E,F,G) and Functional Dependencies
    F={ {A,B}→{C}, { A,C}→{B}, {A,D}→{E}, {B}→{D}, { B,C}→{A},
    {E}→{F}}.
    Check whether the following decomposition of R into
    R1=(A,B,C), R2=(A,C,D,E) and R3=(A,D,F) is satisfying the lossless Decomposition property.
30. What is dependency preservation property for decomposition? Explain why it is important.
31. Give relation schemas for the following normal forms
    i)2NF but not in 3NF ii) 3NF but not in BCNF

The Relational Model – Basic Concepts, Integrity Constraints Over Relations- Key Constraints – Foreign Key Constraints - Relational Algebra Operations - Selection and Projection- Set Operations, Renaming – Joins- Division.
SQL – Various parts of SQL, Basic form of SQL Query, Union, Intersect, and Except, Nested Queries, Aggregate Operators, Null Values, Complex Integrity Constraints in SQL, Triggers.

Relational Model

- The relational model was proposed by Codd in 1970.
- This model is dominant data model and it is foundation for the leading DBMS products like Oracle, DB2, Informix, Sybase, Access, SQL server, MySQL etc.
- The relational model is very simple and elegant with a mathematical basis.
    - A database is a collection of one or more relations, where each relation is a table of rows and columns.
- The major advantages of relational model over older data models are
    - It's simple data representation
    - The ease with which even complex queries can be expressed.
- Laid the foundation for the development of
    - Tuple relational Calculus and then
    - Database standard SQL
- Structured Query Language (SQL) is the most widely used language for creating, manipulating, and queryingrelational DBMS.

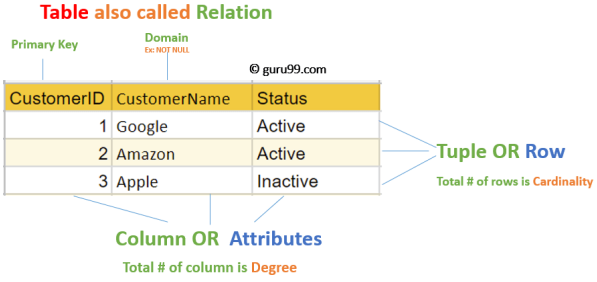
Relation :

- Relation is the main construct for representing data in the relational model.
- A relation consists of
    - Relation schema
    - Relation instance

- ***Relation Schema :***
    - It specifies the relations name, the name of each field, and the domain of each field.
    - The **domain** refers to set of atomic (or indivisible) values – or simply refer it is a data type
    - The domain in Relation Schema is the domain name and it has a set of associated values.Eg:

      Students *(sid : string, name : string, login : string, age : integer, gpa : real)*
    - In this example the relation name is Students.
    - Sid is the field name, which has a domain named string.

- ***Relation Instance:***
    - An instance of a relation is a set of tuples(records).
    - Each tuple has same number of fields as per the Relation Schema.
    - Eg:
    - A Relation Schema specifies the domain of each field in the Relation Instance.
    - Each relation is defined to be a set of unique tuples or rows.



- ***Domain Constraints:***
  - These are the conditions that each instance of the relation satisfies.
  - Domain is nothing but type of that field.
  - The values that appear in a column must be drawn from the domain associated with that column.
  - **Relation Instance should satisfy the domain constraints in the Relation Schema**
- ***Degree:***
  - Degree of a relation is the number of fields in that relation.
  - It is also called as *Arity* of a relation.
  - Eg: Student relation has a degree – 5.
- ***Cardinality of a relation instance :***
  - It is the number of tuples in the relation instance.
  - Eg: student relation instance cardinality is – 6
- ***Relational database :***
  - Collection of relations with distinct names.
- ***Relational Database Schema :***
  - It is the collection of schemas of relations in the database.
- ***Instance of a Relational Database:***
  - Collection of Relational Instances, one per Relation Schema in the Database Schema.
- ***Creating and modifying relations using SQL :***
  - SQL uses the word table to denote the relation.
  - Data Definition Language (DDL) is a subset of SQL and it supports the creation, deletion and modification of tables.
  - The CREATE TABLE statement is used to define a new table.
  - Eg:

```
CREATE  TABLE  Student (sid      VARCHAR2(20),
                        name     VARCHAR2(30),
                        login    VARCHAR2(20),
                        age      NUMBER(3),
                        gpa      NUMBER(5,2) )
```

  - Tuples are inserted using the INSERT command.
  - Eg to insert a single tuple

```
INSERT
INTO Student (sid, name, login, age, gpa)
VALUES (53688, 'smith', 'smith@ee', 18, 3.2)
```

  - If you omit the list of column names, then the values should be placed in appropriate order.
  - Tuples can be deleted using DELETE command.
  - Eg:

```
DELETE
FROM Student
WHERE name = 'Smith'
```

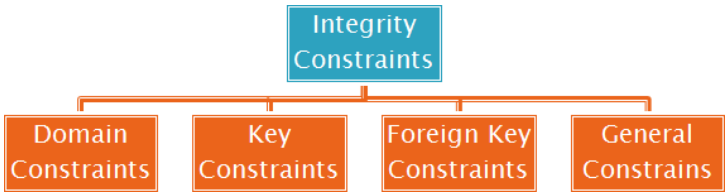
  - UPDATE command is used to modify the column values in an existing row.
  - Eg:

```
UPDATE students
SET age = age + 1, gpa = gpa + 1
WHERE  sid = 53688
```

Integrity Constraints over Relations

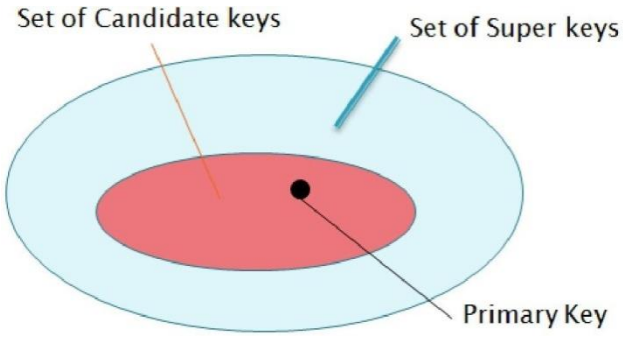
- **Integrity Constraint :** A condition specified on a database schema, which restrict the data that can be stored in the instance of a database.
- If a database instance satisfies all the integrity constraints specified on the database schema, then it is called a legal instance.
- Integrity Constraints (IC) are specified and enforced at different times.
    - At the time of defining a database schema.
    - After defining schema, based on the need of application.

- Different kinds of Integrity Constraints



Key Constraint :

- Key Constraint is a statement that a minimal subset of fields of a relation is a unique identifier for a tuple.
- Super key, candidate key and primary key are mainly used keys. The relation between these three is specifiedin the below picture.



- **Super key :**
    - Super key is the set of fields that contains a key.
    - Every relation is guaranteed to have a key. Because the set of all fields is a super key.

- **Candidate Key :**
    - Candidate key is set of fields that uniquely identify a tuple according to the key constraint .
    - We often call this as simply 'Key'
    - Eg: sid is a candidate key in student relation.
    - In candidate keys,
        1. Two distinct tuples in a legal instance cannot have identical values in all the fields of a key.
        2. No subset of the set of fields in a key is a unique identifier for a tuple.
    - A relation may have several candidate keys.
    - *{ sid }* – this is a candidate key.
    - *{ login, age }* – if this is the key, then the meaning is two students may have same login or age, but notboth.
    - *{ sid, sname }* – it is not a key, because this set properly contains key. This is a super key.

- **Primary key :**
    - A database designer identifies one key out of all candidate keys as a primary key.
    - A tuple can be referred from other relation in the database by storing the values of its

primary keyfields.

- Null values are not allowed to appera in a primary key field.
- Specifying key constraints in SQL
- In SQL, we can declare that a subset of the columns of a table constitute a key using the UNIQUEconstraint.
- One of these candidate keys is declared as primary key using the PRIMARY KEY constraint.

- Eg:

```
CREATE TABLE Students(
        sid             VARCHAR2(20),
        name            VARCHAR2(30),
        login           VARCHAR2(20),
        age             NUMBER(2),
        gpa             NUMBER(5,2),
        UNIQUE(name, age),
        CONSTRAINT StudentsKey PRIMARY KEY(sid)
    )
```

- Constraint name is displayed, if you violate the constraint.

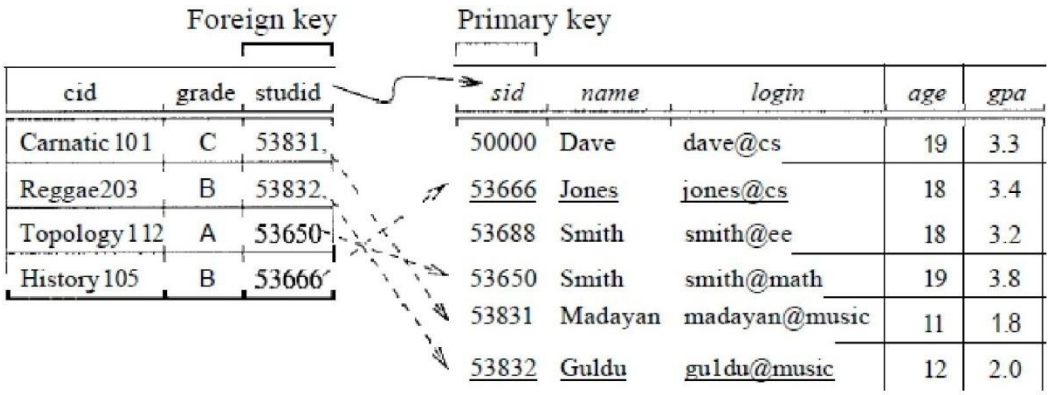
Foreign Key Constraints:

- This is an Integrity Constraint involving two relations.
- For the real world data, the information stored in one relation is linked to the information in the other relation.
- For eg: consider another relation schema.

Enrolled *(Studid : string, cid : string, grade : string)*

This table is used to store grades for the students who are already present in Students relation.

- To ensure the students which are in Students relation should appear in studid field, make studid as foreign key and it has to refer to Students relation.
- **The foreign key in the referencing relation must match the primary key in the referenced relation. i.e., bothshould have same number of columns and compatible types) **



Enrolled (Referencing relation) Students (Referenced relation)

- The insertion of tuple with studid not present in Students relation is rejected by DBMS.
- Sometimes a foreign key could refer to the same relation.
- The appearance of a null in a foreign key does not violate the foreign key constraint.

Specifying foreign key constraints in SQL:

```
CREATE TABLE Enrolled (
        studid          VARCHAR2(20),
        cid             VARCHAR2(20),
```

```
    grade          VARCHAR2(2),
    PRIMARY KEY(studid, cid),
    FOREIGN KEY(studid) REFERENCES Students
  )
```

- ***PRIMARY KEY(studid, cid)*** – allows to put only one grade per studid and cid.
- ***FOREIGN KEY(studid) REFERENCES Students*** – Every studid value must appear in primary key of Studentsrelation.

General Constraints :

- Sometimes, you may require student ages are within a certain range of values (or) a marks column should takevalues within the range 1 to 100.
- If these general conditions are specified as Integrity Constraints, then DBMS rejects the insertions and updating that violate these constraints.
- These ICs are very useful in preventing data entry errors.
- General constraints can be specified in the form of table constraints are assertions.

Table constraints are associated with a single table and checked whenever the table is modified.

- Table constraints have the form

CHECK (conditional – expression)

- Eg: A relation which enforces a restriction on age column. Only the students with age 16 years to 20 years areallowed by the relation.

```
CREATE TABLE Students(
        sid          VARCHAR2(20),
        name         VARCHAR2(30),
        login        VARCHAR2(20),
        age          NUMBER(2),
        gpa          NUMBER(5,2),
        UNIQUE(name, age),
        CONSTRAINT StudentsKey PRIMARY
        KEY(sid),CHECK (age>=16 AND age<=20)
)
```

- Assertions involve several tables and checked whenever any of these tables is modified.

Relation Algebra and Calculus

- These are formal query languages associated with the relational model.
- Query Languages are specialized languages for asking questions / queries that involve data in the database.
- Queries in Relational algebra (RA) are composed using a collection of operators. Each query describes a step bystep procedure for computing the desired answer.
- Query in Relational Calculus describes the desired answer without specifying how the answer is computed. This is non-procedural and this style of querying is called declarative.
- These formal query languages greatly influenced commercial query languages such as SQL.

Sample Queries are presented using the following Schema
Sailors ***(sid : integer, sname : string, rating : integer)***
Boats ***(bid : integer, bname : string, color : string)***
Reserves ***(sid : integer, bid : integer, day : date)***

Instance S1 of Sailors

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Instance S2 of Sailors

sid	sname	rating	age
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	35.0
58	Rusty	10	35.0

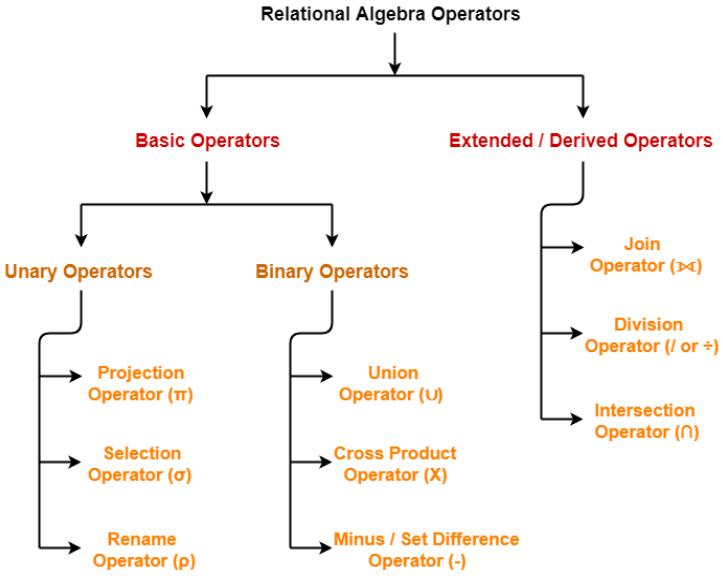
Instance R1 of Reserves

sid	bid	day
22	101	10/10/96
58	103	11/12/96

Relational Algebra

- It is one of the formal query languages associated with relational model.
- Collection of operators is used to compose a query in RA. These operators enable a user to specify basic retrieval requests as relational algebra expressions.
- Every operator accepts relation instances as arguments and returns a relation instance as the result.

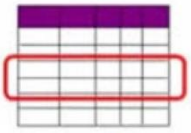
- Complex queries can be easily composed with RA.
- It is procedural i.e., each relational query describes a step-by-step procedure for computing the desired answer, based on the order on which the operators are applied on the query.



- The basic operators of Relational Algebra are
  - Selection
  - Projection
  - Union
  - Cross-Product
  - Difference
- Additional operators can be defined in terms of basic operators.

Selection:

- It allows manipulating data in a single relation.
- It is used to select subset of the tuples from a relation that satisfies a condition.
- The select operation is unary i.e., it is applied to a single relation.
- This operator is applied to each tuple individually.
- It can be visualized as horizontal partition.



- In general, the select operation is denoted as

$$\sigma_{(\text{select condition})}(\text{Relation})$$ where σ is the select operator

<select condition> is the Boolean expression specified on the attributes of the R. It can have the form

<attribute name> <comparison operator> <attribute name>

(or)

<attribute name> <comparison operator> <constant>

These clauses can be connected by standard Boolean operators AND, OR and NOT.

- Eg: Retrieve rows corresponding to expert sailors.

$$\sigma_{\text{rating}>8}(S2)$$

- Eg: Retrieve rows whose age is greater than 40 and rating is less than 9.

$$\sigma_{(\text{age}>40 \text{ AND } \text{rating}<9)}(S2)$$

- The degree of the relation resulting from select operation is same as the degree of R. i.e., for any condition C,

$$|\sigma_C(R)| \leq |R|$$

- Select operation is commutative.

$$\sigma_{(\text{Cond1})}(\sigma_{(\text{cond2})}(R)) = \sigma_{(\text{Cond2})}(\sigma_{(\text{cond1})}(R))$$

- In SQL, the select condition is typically specified in the WHERE clause of a query.
- Eg: for the following query

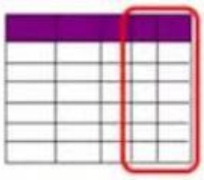
$$\sigma_{(\text{Dno}=4 \text{ AND } \text{salary}>25000)}(\text{EMPLOYEE})$$

- Corresponding SQL query is

```
SELECT *
FROM  EMPLOYEE
WHERE Dno=4 AND Salary>25000
```

Project Operation:

- Project operation selects certain columns from the table and discards the other columns.
- The result of the project operation can be visualized as a vertical partition of the relation into two relations.



- The general form of the PROJECT operation is

$$\pi_{\text{<attribute-list>}}(\text{Relation})$$

- Eg: Find out all sailor names and ratings

$$\pi_{\text{sname,rating}}(S2)$$

- Duplicate tuples may occur in the relation, if the attribute-list includes only non-key attributes. Project operator removes duplicate tuples, so the result is set of distinct tuples, hence the result is a valid relation.
- Eg: Find out only ages of sailors

$$\pi_{\text{age}}(S2)$$

- Commutativity does not hold on project operator.
- moreover

$$\pi_{(List1)}(\pi_{(List2)}(R)) = \pi_{(List1)}(R)$$

- As the result of a RA expression is a relation, we can substitute that expression whenever a relation isexpected.
- Eg: Find out the names and ratings of expert sailors.

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

- In SQL, the project attribute list is specified in the select clause of a query.
- Eg: for the following operation:

$$\pi_{gender,salary}(EMPLOYEE)$$

- The corresponding SQL query is

```
SELECT  DISTINCT  gender, salary FROM EMPLOYEE
```

Rename operation :

- Used to rename the relation name or the attribute name or both.
- It is an unary operator.
- The general forms of RENAME operation when applied to a relation R of degree n is denoted by the followingforms.

$\rho_{S(B_1,...B_n)}(R)$ - S is the new relation name and B1, B2,...Bn are new attribute names.
$\rho_S(R)$ - Renames the relation only.
$\rho_{(B_1,...B_n)}(R)$ - Renames attributes only

- Renaming in SQL is accomplished by aliasing.

```
SELECT E.fname FirstName
    FROM  EMPLOYEE  E
    WHERE  E.dno = 5
```

Set operations:

Standard operations on sets available in RA are

- Union
- Intersection
- Set-difference
- Cross-product

UNION :

- R U S returns a relation instance containing all tuples that occur in either relation instance R or relationinstance S or both.

$$R \cup S = \{ t \mid t \in R \vee t \in S \}$$

- It is a binary operation.
- R and S must be union compatible, and the schema of the result is identical to the schema of R.
- Two relation instances are said to be compatible, if
  - They have the same number of fields, and
  - Corresponding fields, taken in order from left to right, have the same domains.
- Field names are not used in defining union compatibility.

  Eg: S1 U S2

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	Yuppy	9	35.0
44	Guppy	5	35.0

INTERSECTION:

- R ∩ S returns a relation instance containing all tuples that occur in both R and S

$$R \cap S = \{ t \mid t \in R \; \forall t \in S \}$$

- It is a binary operation.
- The relations R and S must be union compatible, and the schema of the result is defined to be identical to theschema of R.
- Eg: R ∩ S

sid	sname	rating	Age
31	Lubber	8	55.5
58	Rusty	10	35.0

SET DIFFERENCE:

- R – S returns a relation instance containing all tuples that occur in R but not in S.

$$R - S = \{ t \mid t \in R \; \forall t \notin S \}$$

- It is also a binary operation.
- The relation R and S must be union compatible and the schema of the result is defined to be identical to the schema of R.
  Eg: R – S

sid	sname	rating	age
22	Dustin	7	45.0

- Union and intersection are commutative operations.

$$R \cup S = S \cup R$$
$$R \cap S = S \cap R$$

- Both union and intersection can be treated as N-ary operations, applicable to any number of relations becauseboth are also associative operations.

$$R \cup (S \cup T) = (R \cup S) \cup T$$
$$R \cap (S \cap T) = (R \cap S) \cap T$$

- Set difference operation is not commutative.

$$R - S \neq S - R$$

- In SQL UNION, INTERSECT, EXCEPT corresponds to set operations ∪, ∩ and −.
- In addition SQL has multiset operations UNION ALL, INTERSECT ALL, EXCEPT ALL, that do not eliminate duplicates.

Cross-Product:

- R × S returns a relation instance whose schema contains all the fields of *R* (in the same order as they appearin R) followed by all the fields of *S* (in the same order as they appear in S).
- The result of R × S contains tuples of the form *<r,s>* for each pair of tuples $r \in R, s \in S$.
- It is also called as Cartesian product.
- If *R* and *S* contains a field with the same name then it creates a naming conflict. These fields are unnamed andare referred to solely by position.

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

S1

sid	bid	day
22	101	10/10/96
58	103	11/12/96

R1

S1 X R1 =

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	22	101	10/10/96
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	22	101	10/10/96
31	lubber	8	55.5	58	103	11/12/96
58	rusty	10	35.0	22	101	10/10/96
58	rusty	10	35.0	58	103	11/12/96

- The relations on which *X* is applied do not have to be union compatible.
- This set operation produces a new element by combining every member (tuple) from one relation with every member (tuple) from other relation.
- Following table illustrates the degree and cardinality of cross product. Let R be relation with n fields and n_R records and S be a relation with m fields and n_S tuples.

	R	S	R X S
Degree	n	m	n + m
Cardinality	n_R	n_S	$n_R * n_S$

Joins:

- Join is defined as a cross product followed by selections and projections.
- It is one of the most useful operations in Relational Algebra.
- It is the most common way to combine information from two or more relations.
- There are several variants of the join operation.
- **Conditional Join:**
  - It is most general version of join.
  - This join accepts a join condition C, and a pair of relation instances as arguments and returns a relationinstance.
  - The form of join condition is similar to selection condition.
  - General form of conditional join is

$$R \bowtie_{(\text{join condition})} S$$

    - Which is similar to

$$R \bowtie_C S = \sigma_C (R \times S)$$

    - $\bowtie$ –cross product followed by selection.

(sid)	sname	rating	age	(sid)	bid	day
22	dustin	7	45.0	58	103	11/12/96
31	lubber	8	55.5	58	103	11/12/96

$$S1 \prec_{S1.sid < R1.sid} R1$$

  - The resultant schema is same as that of cross-product.
  - It is sometimes also called as Theta – join.
  -
- **Equi Join:**
  - It is a special case of join operation where the condition consists of equalities of the form *R.name1 = S.name2*.
  - In equijoin, there is a refinement of additional projection (one field of equalities is dropped)

- The schema of the result of an equijoin contains the fields of R followed by the fields of S that do notappear in the join condition.

sid	sname	rating	age	bid	day
22	dustin	7	45.0	101	10/10/96
58	rusty	10	35.0	103	11/12/96

$$S1 \succ_{sid} R1$$

- **Natural Join :**
  - It is an equijoin on all common fields.
  - It is a special case of equijoin, where we can simply omit the join condition.
  - The default is that the join condition is collection of equalities on all common fields.
  - We can simply denote it as R ⋈ S
  - In R ⋈ S, if they do not have common fields then the result is simply cross product.

Division:

- It is not supported as primitive operator, but useful for expression queries like "*Find sailors who have reserved **all** boats*"
- Let *A(x,y)*, and *B(y)* then the division operation *A/B* is the set of all x values such that for every y value in B,there is a tuple *(x,y)* in *A*.

sno	pno
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

A

pno
p2

B1

sno
s1
s2
s3
s4

A/B1

pno
p2
p4

B2

sno
s1
s4

A/B2

pno
p1
p2
p4

B3

sno
s1

A/B3

- A/Bi gives suppliers who supply all parts listed in Bi

Some more examples :

1. **Find the names of the sailors who reserved boat no. 103**

Solution 1: $\pi_{sname}((\sigma_{bid=103} \text{Reserves}) \succ Sailors)$

Solution 2: $\rho\ (Temp1, \sigma_{bid=103} \text{Reserves})$

$\rho\ (Temp2, Temp1 \succ Sailors)$

$\pi_{sname}(Temp2)$

Solution 3: $\pi_{sname}(\sigma_{bid=103}(\text{Reserves} \succ Sailors))$

We first compute the set of tuples in Reserves with bid = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances R2 and S3, it yields a relation that contains just one field, called sname.

2. **Find the names of sailors who have reserved a red boat.**

❖ Information about boat color only available in Boats; so need an extra join:

$$\pi_{sname}((\sigma_{color='red'}, Boats) \succ \text{Reserves} \succ Sailors)$$

3. **Find the colors of boats reserved by Lubber.**

$$\pi_{color}((\sigma_{sname='Lubber'} Sailors) \bowtie Reserves \bowtie Boats)$$

4. **Find the names of sailors who have reserved at least one boat.**

$$\pi_{sname}(Sailors \bowtie Reserves)$$

The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple `attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same *sid* value, that is, the sailor has made some reservation.

5. **Find the names of sailors who have reserved a red or a green boat.**

$$\rho\ (Tempboats, (\sigma_{color='red' \vee color='green'}, Boats))$$

$$\pi_{sname}(Tempboats \lessdot \text{Reserves} \lessdot Sailors)$$

6. **Find the names of sailors who have reserved a red and a green boat.**

$$\rho\ (Tempred, \pi_{sid}((\sigma_{color='red'}, Boats) \lessdot \text{Reserves}))$$

$$\rho\ (Tempgreen, \pi_{sid}((\sigma_{color='green'}, Boats) \lessdot \text{Reserves}))$$

$$\pi_{sname}((Tempred \cap Tempgreen) \lessdot Sailors)$$

The two temporary relations compute the sids of sailors , and their intersections is the sailors who reserved a red and a green boat

7. **Find the names of sailors who have reserved at least 2 boats.**

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$

$$\rho(Reservationpairs(1 \to sid1, 2 \to sname1, 3 \to bid1, 4 \to sid2,$$

$$5 \to sname2, 6 \to bid2), Reservations \times Reservations)$$

$$\pi_{sname1}\sigma_{(sid1=sid2) \wedge (bid1 \neq bid2)} Reservationpairs$$

First we compute tuples of the form sid,sname,bid, where sailor sid has made a reservation for boat bid; this set of tuples is the temporary relation Reservations. Next we _nd all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: In order to show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances B1, R2, and S3, the sailors with sids 22, 31, and 64 have each reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names Dustin, Horatio, and Lubber.

8. **Find the sids of sailors with age over 20 who have not reserved a red boat.**

$$\pi_{sid}(\sigma_{age>20} Sailors) -$$

$$\pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

This query illustrates the use of the set-difference operator. Again, we use the fact that sid is the key for Sailors. We first identify sailors aged over 20 (over instances B1, R2, and S3, sids 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (sids 22, 31, and 64), to obtain the answer (sids 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their sids (as shown above), and then join with Sailors and project the sname values.

9. **Find the names of sailors who have reserved all boats.**

$$\rho\ (Tempsids, (\pi_{sid,bid} \text{Reserves}) / (\pi_{bid} Boats))$$

$$\pi_{sname}(Tempsids \lessdot Sailors)$$

The intermediate relation Tempsids is defined using division, and computes the set of sids of sailors who have reserved every boat (over instances B1, R2, and S3, this is just sid 22). Notice how we define the two relations that the division operator (/) is applied to—the first relation has the schema (sid,bid) and the

second has the schema (bid). Division then returns all sids such that there is a tuple 〈 sid,bid 〉 in the first relation for each bid in the second. Joining Tempsids with Sailors is necessary to associate names with the selected sids; for sailor 22, the name is Dustin.

10. Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute only bids of boats named Interlake in defining the second argument to the division operator. Over instances B1, R2, and S3, Tempsids evaluates to sids 22 and 64, and the answer contains their names, Dustin and Horatio.

Structured Query Language (SQL)

The basic form of an SQL query is as follows:

SELECT **[DISTINCT]** select-list
FROM from-list
WHERE qualification

Such a query intuitively corresponds to a relational algebra expression involving selections, projections, and cross-products. Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a crossproduct of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause. Let us consider a simple query.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance $S3$ of Sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Figure 4.16 An Instance $R2$ of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Figure 4.17 An Instance $B1$ of Boats

✓ **Find the names and ages of all sailors.**

```
SELECT DISTINCT S.sname, S.age
FROM Sailors S
```

The answer is a set of rows, each of which is a pair <sname, age >. If two or more sailors have the same name and age, the answer still contains just one pair with that name and age. This query is equivalent to applying the projection operator of relational algebra. The answer to this query with and without the keyword DISTINCT on instance S3 of Sailors is shown in Figures. The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35

SNAME	AGE
Dustin	45
Brutus	33
Lubber	55.5
Andy	25.5
Rusty	35
Horatio	35
Zorba	16
Horatio	35
Art	25.5
Bob	63.5

SNAME	AGE
Andy	25.5
Zorba	16
Art	25.5
Bob	63.5
Lubber	55.5
Horatio	35
Dustin	45
Brutus	33
Rusty	35

- **Find all sailors with a rating above 7**

```
SELECT    S.sid, S.sname, S.rating, S.age
FROM      Sailors S
WHERE     S.rating > 7
```

SID	SNAME	RATING	AGE
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35
64	Horatio	9	35
71	Zorba	10	16
74	Horatio	9	35

We now consider the syntax of a basic SQL query in more detail.

- The from-list in the FROM clause is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The select-list is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the WHERE clause is a boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<, <=, = , <>, >=, >}. An expression is a column name, a constant, or an (arithmetic or string) expression.
- The DISTINCT keyword is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated. the syntax of a basic SQL query, they don't tell us the meaning of a query. The answer to a query is itself a relation which is a multiset of rows in SQL whose contents can be understood by considering the following conceptual evaluation strategy:
  - Compute the cross-product of the tables in the from-list.
  - Delete those rows in the cross-product that fail the qualification conditions.
  - Delete all columns that do not appear in the select-list.
  - If DISTINCT is specified, eliminate duplicate rows.

✓ **Find the names of sailors who have reserved boat number 103.**

It can be expressed in SQL as follows.

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid AND R.bid=103
```

✓ **Find the sids of sailors who have reserved a red boat.**

```
SELECT R.sid
FROM Boats B, Reserves R
WHERE B.bid = R.bid AND B.color = 'red'
```

✓ **Find the names of sailors who have reserved a red boat.**

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'.
```

✓ **Find the colors of boats reserved by Lubber.**

```
SELECT B.color
 FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid
AND R.bid = B.bid AND S.sname = 'Lubber'
```

✓ **Find the names of sailors who have reserved at least one boat.**

```
SELECT S.sname
FROM Sailors S, Reserves R
WHERE S.sid = R.sid
```

✓ *Compute increments for the ratings of persons who have sailed two different boats on the same day.*

```
SELECT S.sname, S.rating+1 AS rating
FROM Sailors S, Reserves R1, Reserves R2
WHERE S.sid = R1.sid AND S.sid = R2.sid
AND R1.day = R2.day AND R1.bid <> R2.bid
```

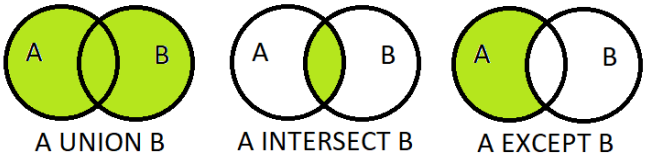
✓ *Find the ages of sailors whose name begins and ends with B and has at least three characters.*

```
SELECT S.age
FROM Sailors S
WHERE S.sname LIKE 'B %B'
```

UNION, INTERSECT, AND EXCEPT

SQL provides three set-manipulation constructs that extend the basic query form presented earlier. Since the answer to a query is a multiset of rows, it is natural to consider the use of operations such as union, intersection, and difference. SQL supports these operations under the names UNION, INTERSECT, and EXCEPT.

SQL also provides other set operations:
IN (to check if an element is in a given set), op ANY, op ALL(to compare a value with the elements in a given set, using comparison operator op), and EXISTS
(to check if a set is empty). IN and EXISTS can be prefixed by NOT, with the obvious modification to their meaning. We cover UNION, INTERSECT, and EXCEPT in this section.



Consider the following query:

✓ *Find the names of sailors who have reserved both a red and a green boat.*

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
UNION
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

✓ *Find the names of sailors who have reserved a red or a green boat.*

```
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid
AND R.bid = B.bid
AND (B.color = 'red' OR B.color = 'green')
```

✓ *Find the sids of all sailors who have reserved red boats but not green boats.*

```
SELECT S.sid
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S2.sid FROM Sailors S2, Reserves R2, Boats B2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'
```

Note that UNION, INTERSECT, and EXCEPT can be used on *any* two tables that are union-compatible, that is, have the same number of columns and the columns, taken in order, have the same types.

NESTED QUERIES

A nested query is a query that has another query embedded within it; the embedded query is called a subquery.

✓ **Find the names of sailors who have reserved boat 103.**

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
                 FROM Reserves R
                 WHERE R.bid = 103 )
```

The nested subquery computes the (multi)set of *sid*s for sailors who have reserved boat 103, and the top-level query retrieves the names of sailors whose *sid* is in this set. The IN operator allows us to test whether a value is in a given set of elements; an SQL query is used to generate the set to be tested.

✓ **Find the names of sailors who have reserved a red boat.**

```
SELECT S.sname
FROM Sailors S
WHERE S.sid IN ( SELECT R.sid
                 FROM Reserves R
                 WHERE R.bid IN ( SELECT B.bid
                                  FROM Boats B
                                  WHERE B.color = 'red' )
```

The innermost subquery finds the set of bids of red boats. The subquery one level above finds the set of sids of sailors who have reserved one of these boats.

✓ **Find the names of sailors who have not reserved a red boat.**

```
SELECT S.sname
FROM Sailors S
WHERE S.sid NOT IN ( SELECT R.sid
                     FROM Reserves R
                     WHERE R.bid IN ( SELECT B.bid
                                      FROM Boats B
                                      WHERE B.color = 'red' )
```

Correlated Nested Queries

In the nested queries that we have seen thus far, the inner subquery has been completely independent of the outer query. In general the inner subquery could depend on the row that is currently being examined in the outer query

✓ **Find the names of sailors who have reserved boat number 103.**

```
SELECT S.sname
FROM Sailors S
WHERE EXISTS ( SELECT *
               FROM Reserves R
               WHERE R.bid = 103 AND R.sid = S.sid )
```

The EXISTS operator is another set comparison operator, such as IN. It allows us to test whether a set is nonempty. Thus, for each Sailor row S, we test whether the set of Reserves rows R such that R.bid = 103 AND S.sid = R.sid is nonempty. If so, sailor S has reserved boat 103, and we retrieve the name. The subquery clearly depends on the current row S and must be re-evaluated for each row in Sailors. The occurrence of S in the subquery (in the form of the literal S.sid) is called a correlation, and such queries are called correlated queries.

Set-Comparison Operators

We have already seen the set-comparison operators EXISTS, IN, and UNIQUE, along with their negated versions. SQL also supports op ANY and op ALL, where op is one of the arithmetic comparison operators {<, <=, =, <>, >=, >}. (SOME is also available, but it is just a synonym for ANY.)

✓ ***Find the sailors with the highest rating .***

```
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating FROM Sailors S2 )
```

The subquery computes the set of all rating values in Sailors. The outer WHERE condition is satisfied only when *S.rating* is greater than or equal to each of these rating values, i.e., when it is the largest rating value.

✓ **Find the names of sailors who have reserved all boats.**

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid
                    FROM Boats B )
EXCEPT
(SELECT R.bid
FROM Reserves R
WHERE R.sid = S.sid ))
```

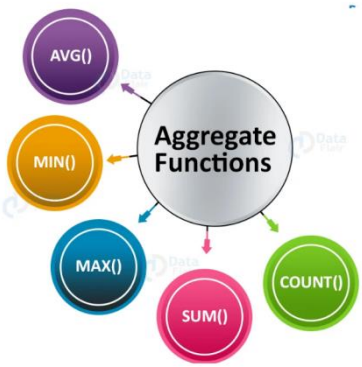
✓ **Find sailors whose rating is better than some sailor called Horatio.**

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating
                       FROM Sailors S2
                       WHERE S2.sname = 'Horatio' )
```

AGGREGATE OPERATORS

We now consider a powerful class of constructs for computing *aggregate values* such as MIN and SUM. These features represent a significant extension of relational algebra. SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. COUNT ([DISTINCT] A): The number of (unique) values in the A column.
2. SUM ([DISTINCT] A): The sum of all (unique) values in the A column.
3. AVG ([DISTINCT] A): The average of all (unique) values in the A column.
4. MAX (A): The maximum value in the A column.
5. MIN (A): The minimum value in the A column.



✓ **Find the average age of all sailors.**

```
SELECT AVG (S.age)
FROM Sailors S
```

✓ **Find the average age of sailors with a rating of 10.**

```
SELECT AVG (S.age)
FROM Sailors S
WHERE S.rating = 10
```

✓ ***Find the name and age of the oldest sailor.***

```
SELECT S.sname, S.age
FROM Sailors S
WHERE S.age = ( SELECT MAX (S2.age)
                FROM Sailors S2 )
```

The GROUP BY and HAVING Clauses

we want to apply aggregate operations to each of a number of groups of rows in a relation, where the number of groups depends on the relation instance (i.e., is not known in advance).

- **GROUP BY** clause is used to arrange identical data into groups
- **HAVING** clause can be used to specify conditions on groups.
- The general form of an SQL query with GROUP BY and HAVING extensions is

```
SELECT      column-names
FROM        table-name list
```

```
WHERE       condition
GROUP BY    grouping list of columns
HAVING      condition
```

- Important points
  - The select-list consists of
    1. A list of column names
    2. A list of terms having aggregate operators
  - Every column that appear in select list must appear in grouping list, as each row in the result corresponds to one group.
  - HAVING clause determines whether an answer row is to be generated for a given group or not.
    1. HAVING clause must have single value per group.
    2. HAVING requires that a GROUP BY clause is present.
    3. WHERE and HAVING can be in the same query.
  - If GROUP BY is omitted, the entire table is regarded as a single table.

For example, consider the following query.

✓ ***Find the age of the youngest sailor for each rating level.***

```
SELECT S.rating, MIN (S.age)
FROM Sailors S
GROUP BY S.rating
```

✓ ***Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.***

```
SELECT S.rating, MIN (S.age) AS minage
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

✓ ***For each red boat, find the number of reservations for this boat.***

```
SELECT B.bid, COUNT (*) AS sailorcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
```

✓ ***Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.***

```
SELECT S.rating, AVG ( S.age ) AS avgage
FROM Sailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
             FROM Sailors S2
             WHERE S.rating = S2.rating)
```

✓ ***Find those ratings for which the average age of sailors is the minimum over all ratings.***

```
SELECT S.rating
FROM Sailors S
WHERE AVG (S.age) = ( SELECT MIN (AVG (S2.age))
                      FROM Sailors S2
                      GROUP BY S2.rating )
```

✓ ***Find the average age of sailors for each rating level that has at least two sailors.***

```
SELECT S.rating, AVG (S.age) AS avgage
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
          (or)
SELECT S.rating, AVG ( S.age ) AS avgage
```

```
FROM Sailors S
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
             FROM Sailors S2
             WHERE S.rating = S2.rating )
```

NULL Values

- SQL provides a special column value called null.
- We use null when column value is either unknown or inapplicable.
- Presence of null values complicates many issues
- Impact of null values on SQL
  - Comparisons using Null values
    - If one of the operand is null in comparison, then the result is unknown.
    - SQL also provides a special comparison operator IS NULL to test whether a column value is null
  - Logical Connectives
    - Define the Logical operators AND, OR, NOT using three-valued logic (true, false, unknown) when we have null values
    - NOT unknown is unknown
    - OR of two arguments evaluates to true if either argument evaluates to true, and to unknown if one argument evaluates to false and the other evaluates to unknown.
    - AND of two arguments evaluates to false if either argument evaluates to false, and to unknown if one argument evaluates to unknown and the other evaluates to true or unknown.
  - On SQL constructs
    - Where clause eliminates rows for which the condition does not evaluate to true
    - Eliminating rows that evaluate to unknown has significant impact on nested queries involving EXISTS or UNIQUE.
    - Arithmetic operations return null, if one of their argument is null.
    - Aggregate operations simply discard null values
- Disallowing null values
  - Specify **NOT NULL** as part of the field definition.
  - Eg: Sname CHAR(20) NOT NULL
  - The fields in a primary key are not allowed to take on null values.

Joins

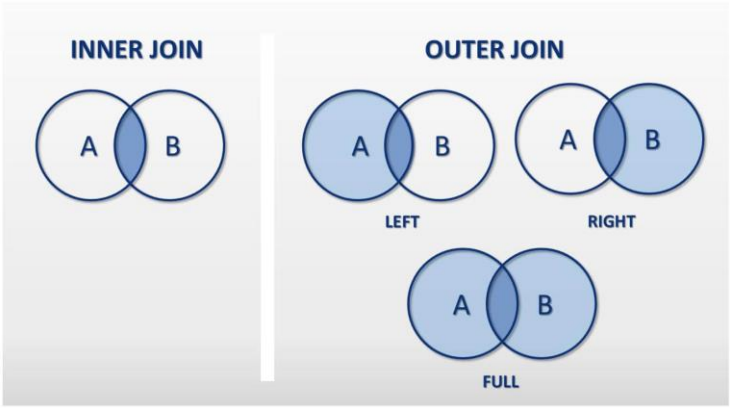
- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Table A

CODE
A
C
D
F

Table B

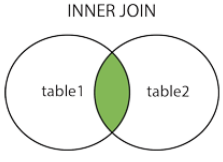
CODE
A
X
C



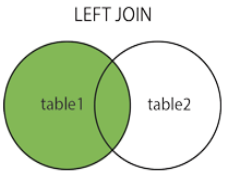
- The Join operations are
  - **INNER JOIN operation**
    - Specifies a join between two tables with an explicit join clause.
    - Syntax:

```
SELECT column_name(s)
FROM table1
INNER JOIN table2
ON table1.column_name = table2.column_name;
```

CODE	CODE
A	A
C	C



– **LEFT OUTER JOIN operation**



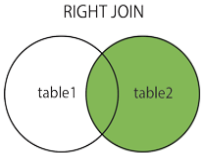
- Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the first table.
- Syntax:

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

- Example

CODE	CODE
A	A
C	C
F	
D	

– **RIGHT OUTER JOIN operation**



- Specifies a join between two tables with an explicit join clause, preserving unmatched rows from the second table.
- Syntax:

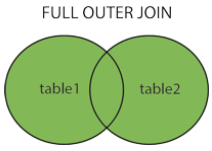
```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

- Example:

CODE	CODE
A	A
C	C
	X

– **FULL OUTER JOIN operation**

- The FULL OUTER JOIN keyword returns all records when there is a match in left (table1) or right (table2) table records.
- Syntax:




```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name = table2.column_name
WHERE condition;
```

- Example:

CODE	CODE
A	A
C	C
F	
D	
	X

Complex integrity constraints in SQL

Constraints over a Single table

- Constraints over a single table can be specified as using table constraints

  CHECK conditional-expression
- When a row is inserted or modified, the conditional expression in the CHECK constraint is evaluated.. If it evaluates to FALSE, the command is rejected.
- Eg: to ensure that the rating must be an integer in the range 1 to 10

```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK (rating >= 1 AND rating <= 10 ))
```

- **To enforce the constraint that Interlake boats cannot be reserved.**

```
CREATE TABLE Reserves (sid    INTEGER,
                       bid    INTEGER,
                       day    DATE,
                       FOREIGN KEY (sid) REFERENCES Sailors
                       FOREIGN KEY (bid) REFERENCES Boats
                       CONSTRAINT noInterlakeRes
                       CHECK ( 'Interlake' <>
                               ( SELECT B.bname
                                 FROM   Boats B
                                 WHERE  B.bid = Reserves.bid )))
```

Domain Constraints and Distinct Types

- A user can create a new domain

```
CREATE DOMAIN ratingval INTEGER DEFAULT 1
                        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

  - Integer is the source data type.
  - New domain type is 'ratingval'
  - Values in 'ratingval' are further restricted by using a CHECK constraint.
  - The Keyword VALUE refers to a value in the domain.
  - The DEFAULT keyword is optional, it is used to associate a default value with a domain.
  - Once a domain is defined, it can be used to restrict column values in a table.

Assertions : ICs over several tables

- Table constraints are associated with a single table.
- The conditional expression in the CHECK clause can refer to other tables.
- Assertions are the constraints which are associated with more than one table.
- Eg: to enforce the constraint – "that the number of boats plus the number of sailors should be less than 100"

```
CREATE ASSERTION smallClub
CHECK (( SELECT COUNT (S.sid) FROM Sailors S )
       + ( SELECT COUNT (B.bid) FROM Boats B)
       < 100 )
```

Note:

- A column constraint imposes a condition on a column in a table.
- A table constraint puts a specified constraint on an entire table.
- An assertion is a constraint that can affect more than one table.

Triggers and Active Databases

- A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes
- A database that has a set of associated triggers is called an active database.
- Parts of a trigger description
  - Event – a change to the database that activates the trigger
  - Condition – a query or test that is run when the trigger is activated.
  - Action – a procedure that is executed when the trigger is activated and its condition is true.

The syntax for creating a trigger is

```
CREATE [OR REPLACE ] TRIGGER trigger_name
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF col_name]
ON table_name
[REFERENCING OLD AS o NEW AS n]
[FOR EACH ROW]
WHEN (condition)
DECLARE
   Declaration-statements
BEGIN
   Executable-statements
EXCEPTION
   Exception-handling-statements
END;
```

Example:

```
CREATE TRIGGER  init_count
BEFORE INSERT ON Students   /*Event*/
DECLARE
        count  INTEGER;
BEGIN
        count := 0;     /* Action */
END;
```

- **Count the no. of students whose age > 18**

```
CREATE TRIGGER  incr_count
AFTER INSERT ON Students          /*Event*/
WHEN (new. age < 18)              /*Condition*/
FOR EACH ROW
BEGIN
    count := count+ 1;            /* Action */
END;
```

Trigger will display the salary difference between the old values and new values

```
CREATE OR REPLACE TRIGGER print_salary_changes
  after  UPDATE ON employees
  FOR EACH ROW
DECLARE
  sal_diff  NUMBER;
BEGIN
  sal_diff  := :NEW.salary  - :OLD.salary;
  DBMS_OUTPUT.PUT(:NEW.ENAME || ': ');
  DBMS_OUTPUT.PUT('Old salary = ' || :OLD.salary || ', ');
  DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
  DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);
END;
/
```

```
update employee
 set salary= salary*1.1
```

Execute | Load Script | Save Script | Cancel

```
VENKAT: Old salary = 132000, New salary = 145200, Difference: 13200
VENKAT1: Old salary = 132000, New salary = 145200, Difference: 13200
: Old salary = 979, New salary = 1076.9, Difference: 97.9
Nirmala: Old salary = 22000, New salary = 24200, Difference: 2200
Pradeep: Old salary = 33000, New salary = 36300, Difference: 3300
Krishna: Old salary = 18700, New salary = 20570, Difference: 1870
6 rows updated.
```

Types of Triggers

- Row-level triggers and Statement-level triggers
- BEFORE and AFTER triggers
- INSTEAD OF triggers
- Triggers on System events and user events.
- **Row Triggers**
  - A **row trigger** is fired each time the table is affected by the triggering statement.
  - For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement. If a triggering statement affects no rows, a row trigger is not run.
- **Statement Triggers**
  - A statement trigger is fired once on behalf of the triggering statement, regardless of the number of rows in the table that the triggering statement affects, even if no rows are affected.
  - For example, if a DELETE statement deletes several rows from a table, a statement-

level DELETE trigger is fired only once.

Example 1: Monitoring Statement Events

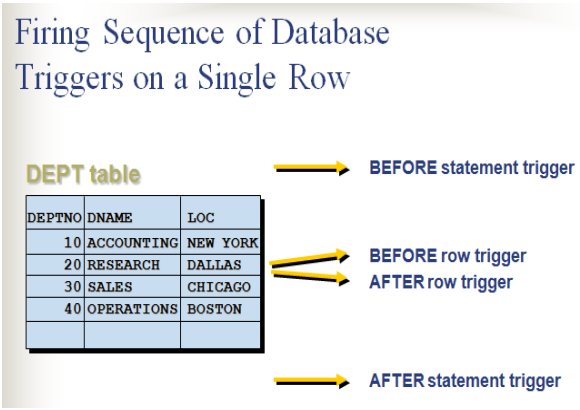
```
SQL> INSERT INTO dept (deptno, dname, loc)
  2  VALUES (50, 'EDUCATION', 'NEW YORK');
```

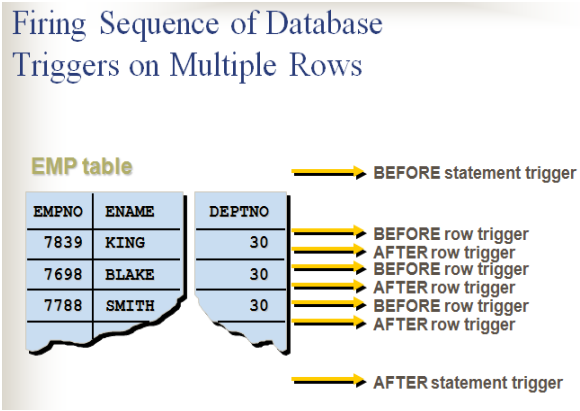
Execute only once even if multiple rows affected

Example 2: Monitoring Row Events

```
SQL> UPDATE emp
  2  SET sal = sal * 1.1
  3  WHERE deptno = 30;
```

Execute for each row of the table affected by the event





- **BEFORE Triggers**
  - BEFORE triggers run the trigger action before the triggering statement is run.
  - This type of trigger is commonly used
    - When the trigger action determines whether the triggering statement should be allowed to complete.
    - To derive specific column values before completing a triggering INSERT or UPDATE statement.
- **AFTER Triggers**
  - AFTER triggers run the trigger action after the triggering statement is run

- **INSTEAD OF TRIGGERS**
  - INSTEAD OF triggers provide a transparent way of modifying views that cannot be modified directly through DML statements.
  - These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle fires the trigger instead of executing the triggering statement.
  - You can write normal INSERT, UPDATE, and DELETE statements against the view and the INSTEAD OF trigger is fired to update the underlying tables appropriately.
  - INSTEAD OF triggers are activated for each row of the view that gets modified.

Triggers on System Events and User Events

- These database events include:
- System events
  - Database startup and shutdown
  - Data Guard role transitions
  - Server error message events
- User events
  - User logon and logoff
  - DDL statements (CREATE, ALTER, and DROP)
  - DML statements (INSERT, DELETE, and UPDATE)
- Triggers on system events can be defined at the database level or schema level.

Restrictions for Database Triggers

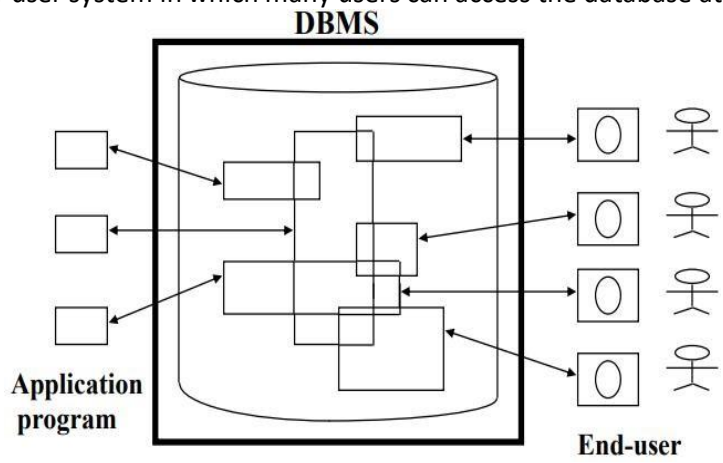
- **Problem:** impossible to determine certain values during execution of a sequence of operations belonging to one and the same transaction
- **Mutating tables:** contain rows which change their values after certain operation and which are used again before the current transaction commits
- **Preventing table mutation:**
  - Should not contain rows which are constrained by rows from other changing tables
  - Should not contain rows which are updated and read in one and the same operation
  - Should not contain rows which are updated and read via other operations during the same transaction

Introduction to Database Management System, Data Independence- Relation Systems and Others, Database system architecture, Introduction- The Three Levels of Architecture-The External Level- the Conceptual Level- the Internal Level- Mapping- the Database Administrator, Various Data Models

The ER Model - The Relational Model, Relational Calculus, Introduction to Database Design, Database Design and ER Diagrams-Entities Attributes, and Entity Sets-Relationship and Relationship Sets - Conceptual Design with ER Model

What is Database System

- A database system is basically a computerized record keeping system. i.e., it is a computerized system whoseoverall purpose is to store information and to allow users to retrieve and update that information on demand.
- A simplified picture of the database system is shown below.
- This system is a multi-user system in which many users can access the database at the same time.



- This system involves four major components.
    1. Data,
    2. hardware,
    3. software and
    4. users.

Data:

- The data in the database will be both integrated and shared.
- *Integrated* : Database is unification of several distinct files, with any redundancies among these files will be partly or wholly removed.
- *Shared* : The database can be shared among different users.

Hardware:

- The hardware components of the system include disk, CPU, main memory etc.

Software:

- Between the physical database and the user is a layer of software, known as Database Management System(DBMS)
- DBMS handles all requests for access to the database.
- **Functions of DBMS:**

    - Shields the database users from hardware level details.
    - Data Dictionary Management
    - Data Storage Management
    - Data Transformation and Presentation
    - Security Management

- Backup and Recovery Management
- Data Integrity Management
- Transaction Management

Users:

- For small databases typically one person defines, construct, and manipulates the data. But for largeorganizations, many people will be involved.
- There are various types of users based on the level of access and how much permission he has toalter/modify/or to any other operation on the database

Different types of users are

1. ***Application programmers***
   - Application Programmers are responsible for writing application programs that use the database. These programs could be written in General Purpose Programming languages such as Visual Basic, Developer, C, FORTRAN, COBOL etc. to manipulate the database. These application programs operate on the data to perform various operations such as retaining information, creating new information, deleting or changing existing information.
2. ***End users***
   - The database primarily exists for their use.
   - End users are those whose jobs require access to the database for querying, updating, and generatingreports.
   - There are several categories of end users:
   - ***Casual end users :*** Needs different information each time. They use a sophisticated database query interface to specify their requests.
   - ***Naive users :*** Any user who does not have any knowledge about database can be in this category.There task is to just use the developed application and get the desired results. For example: Clerical
     staff in any bank is a naïve user. They don't have any dbms knowledge but they still use the database and perform their given task.
   - ***Sophisticated end users :*** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with facilities of the DBMS
3. ***Database Designers:***
   - Responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data.
   - Understands the requirements of the user. The design should be capable of supporting the requirements of all user groups.
4. ***Database administrator (DBA)***
   - The function of managing and maintaining database management systems (DBMS) software
   - He is the chief administrator to oversee and manage the resources.
   - The primary resource is the database itself and the secondary resource is the DBMS and the related softwares.
   - Various *responsibilities of DBA* are

1. ***Policy formulation and implementation***
   a) **Access privileges** - users should access the database only in ways in which they are entitled.
   b) **Security** - Access restrictions ensure that the database is secure. Passwords, encryption, and views implement security. Effective password protection is important.
   c) **Disaster Planning** – Databases can be harmed from hardware and software malfunctions, and outside forces like floods and power outages. DBA's must take active role in formulating disaster recovery plans.
   d) **Archives** – An archive is a place where corporate data is kept. Information in an archive is removed from the database and stored in the archive for future reference. Archives are usually kept in a mass-storage device like a disk, tape, CD, or a DVD. It is important such archives be kept off-site to allow recovery should disaster strike

2. ***Data Dictionary management***
   i. Data Dictionary is like a catalog, it contains a wider range of information, including informationon tables, fields, indexes, and programs.
   ii. The DBA manages and updates the data dictionary, which establishes naming conventions fortables, fields etc., and data integrity rules.
3. ***Training***
   i. DBAgives training on the DBMS and how to access the database.
4. ***DBMS support***
   a)The DBA is charged with all aspects of a DBMS, including selection and management.
5. ***Database Design***
   i. DBA is responsible for tuning the design, i.e., making changes that improve system performance

Functions of DBA in simple terms
• **Authorizing access to the database**
• **Coordinating and monitoring its use.**
• **Acquiring hardware and software resources as needed.**
• **Accountable for problems such as security breaches and poor system response time.**
• **Defining external/internal/conceptual schema**
• **Relationship with users**
• **Defining security and integrity checks**
• **Defining Backup and recovery procedures**
• **Monitoring performance and changing requirements.**

What is a Database?

- A database is collection of related data
  - ***Def***: *A database is a collection of persistent data that is used by the application systems of some given enterprise.*
    - Database is persistent because once it has been accepted by the DBMS for entry, it can subsequentlybe removed from the database only by some explicit request to the DBMS.
    - Enterprise can be commercial, scientific, technical or other organization. It can be a single individual ora complete corporation.

  - A real world scenario will be translated into a database. It is combination of entities, relationships and properties.
  - Entity is any real world object w.r.t our problem domain. It is an object about which we wish to recordinformation.
  - An entity set is a set of that share common properties of entities
  - A relationship is an association among two or more entities or other relationships.
  - The properties/attributes are the information we want to record about entities.
  - Properties can be either simple or complex.

Data and Data Model:

- A data model is a model of the persistent data of some particular enterprise.
- Data models are collection of conceptual tools used to describe data, data relationships and consistencyconstraints.
- Data models are used to describe data stored in database.
- A data model is an abstract, self-contained logical definition of the objects, operators that constitute the abstract machine with which user interacts.
- The objects allow us to model the structure of the data.
- The operators allow us to model its behaviour
- An implementation of the given model is a physical realization on a real machine.

- Model – is what users have to know about
- Implementation – is what users do not have to know about.
- There are number of data models.

The various data models are

- ER model
- Relational model
- Object based data model
- Network data model
- Hierarchical data model.

- The DBMS and database must adhere to a data model.

- Relational model proposed by Codd (Codd, 1970)
  - Everything is a relation
  - Query consists of algebraic composition of a few powerful operators
  - Equivalent to first-order relational calculus
  - Because of its simplicity, it found many supporters (specially at universities)

Why database?

- A database is a collection of data or information which is held together in an organised or logical way.
- Databases can either be paper based or computerised.

The advantages of database system over traditional paper-based methods of record keeping are

1. ***Compactness :*** There is no need for paper files which are big at volume.
2. ***Speed :*** machine retrieves and updates data faster than machine.
3. ***Less drudgery :*** mechanical works like maintaining files by hand is eliminated.
4. ***accurate :*** Accurate, up-to-date information is available on demand at anytime.
5. ***Protection :*** Database can be better protected against unintentional loss & lawful access.
6. Multi-user environment can be provided.

Computerised databases can either use *a File management System* or *Database Management system.*

File Management Systems:

- Before the advent of DBMS, data used to be managed by File processing system. In this data is stored in operating system files.
- This system needs number of application program to allow user to manipulate the information.
- Keeping information in a file processing system has a number of major disadvantages.

They are

1. ***Data redundancy and inconsistency :*** Redundancy means multiple copies of the same data. To access data different programmers create files with different structures in different programming languages. It leads to data duplication.
   Eg: If the student address is stored in several files , and we change only one copy then it leads to data inconsistency.
   Data inconsistency means, multiple copies of the same data may no longer agree.
2. ***Difficulty in accessing the data:*** FPS doesn't allow required data to be retrieved in a convenient and efficient manner.
   Eg: if a user asks a list of customers living in a particular town then they have to write a program manually, which is a tedious job.
3. ***Data isolation:*** New application is needed to retrieve data, as data scatters in multiple files, which has different formats. Writing new applications every time is a difficult task.

4. ***Integrity problems :*** Data stored in the system has to satisfy certain consistency constraints.Eg: minimum balance of an account be Rs.500/-
   We say that data is maintaining integrity, when all these conditions are met.
   In FPS, when new constraints are added, it is difficult to change the programs to enforce them.
5. ***Atomicity Problems :*** Atomicity means either all actions of a program should be executed, or none of theactions should be executed.
   Eg: A fund transfer of Rs. 500/- from Account A to B

	A	B
Assume the initial balances	1000	2000
	A	B
If all actions are done	500	2500
	A	B
If none of them are done	1000	2000
	A	B
But partial execution of operations is not acceptable	500	2000

   It is difficult to ensure atomicity in FPS.

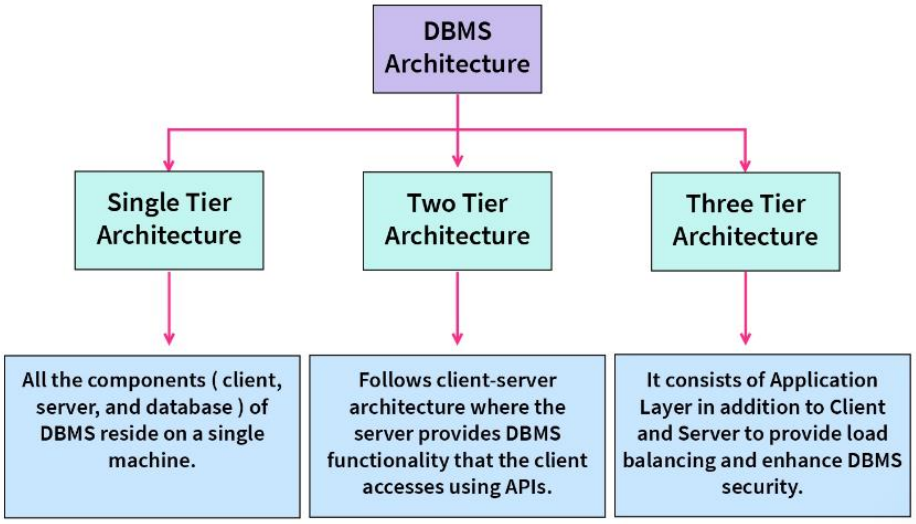
6. ***Concurrent access anomalies :*** When many programs are executed simultaneously, sometimes anomalies maycome. Dealing these anomalies is difficult case of FPS.
7. ***Security Problems :*** Security can be provided by keeping passwords to files. But this is not sufficient when we have to allow users to access different subsets of the data.

These problems of FPS prompted the development of database systems. A DBMS is a piece of software designed to make the preceding tasks easier.

Benefits of the Database Approach:

1. ***Data Independence :*** The DBMS provides an abstract view of data that hides data representation and storagedetails from application programs.
2. ***Efficient data access :*** DBMS uses good techniques to store and retrieve data efficiently.
3. ***The data can be shared :*** Existing applications can share the data in the database and also new applications can be developed to operate on the same data.
4. ***Redundancy can be reduced :*** Every application in non-database system has its own private files, which leads to redundancy in stored data. This redundancy wastes the storage and causes inconsistencies. It can be eliminated in database systems as there is centralized control of data.
5. ***Inconsistency can be avoided (to some extent) :*** Data is said to be inconsistent if the duplicate entries will not agree. When one of the two entries is updated and other not then the database is said to be inconsistent state. Inconsistency can be avoided by representing the fact by a single entry in databases.
6. ***Transaction support can be provided :*** A transaction is a logical unit of work, typically involving several database operations. Atomicity can be achieved by transaction support.
7. ***Integrity can be maintained :*** DBMS can enforce integrity constraints easily. For eg: before inserting salary information for an employee, the DBMS can check that the department budget is not exceeded.
8. ***Security can be enforced :*** Ensuring access to the database through proper channel is possible in DBMS. DBA defines security constraints to be checked whenever access is attempted to sensitive data.
9. ***Concurrent access and crash recovery :*** A DBMS provides concurrent access to the data in such a manner that users can think of the data as being accessed by only one user at a time.
10. ***Reduced Application Development Time:*** As DBMS supports important functions common to many applications, developing new applications can be done in reduced time.

Types of DBMS Architecture:



1. Single Tier Architecture

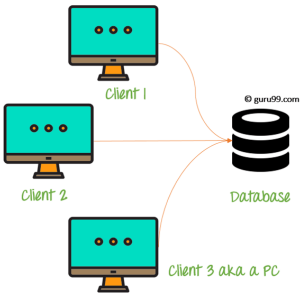
Single Tier DBMS Architecture is the most straightforward DBMS architecture. All the DBMS components reside on a single server or platform, i.e., the database is directly accessible by the end-user. Because of this direct connection, the DBMS provides a rapid response, due to which programmers widely use this architecture to enhance the local application.

In this structure, any modifications done by the client are reflected directly in the database, and all the processing is done on a single server. Also, no network connection is required to perform actions on the database. This database management system is also known as the **local database system**.

2. Two Tier Architecture

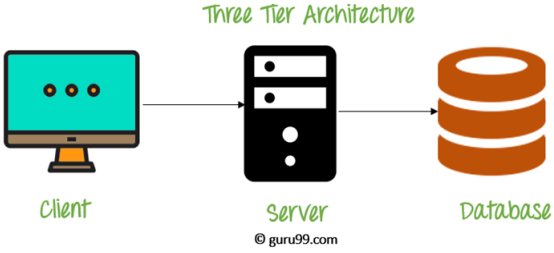
The application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

- It provides added security to the DBMS as it is not exposed to the end-user directly.
- It also provides direct and faster communication.



3. Three Tier Architecture

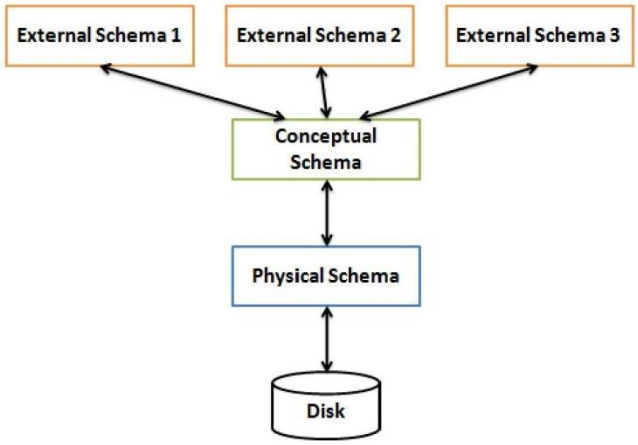
The client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients. Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.



The three levels of the Architecture

Data Abstraction:

- Abstraction means hiding the complexity and providing only necessary details. Data Abstraction hides theirrelevant details from the users.
- Database system comprises of complex data structures. Developers use abstraction in order to
  - Make the system efficient in terms of retrieval of data
  - Reduce complexity in terms of usability of users.
- This approach simplifies the database design.
- DBMS provides an abstract view of the data stored in the database i.e., the system hides certain details of howthe data are stored and maintained. To retrieve data efficiently, DBMS uses complex data structures. Since many database users are not computer trained, developers hide the complexity from users through several levels of abstraction.
- The Data in a DBMS is described at three levels of abstraction.
- The database description consists of a schema at each of these three levels of abstraction. They are
  - Conceptual schema
  - Physical schema
  - External schema
- Levels of abstraction are shown in below figure.



- Conceptual schema (logical schema) describes the stored data interms of the data model of the DBMS.
- Physical schema specifies storage details
- External schema allows access to individual users or group of users.

Architecture of DBMS

The Architecture of most of commercial DBMS available today is mostly based on this ANSI-SPARC[1] database architecture.

The **objectives** of this three-level architecture is to separate the user's view,

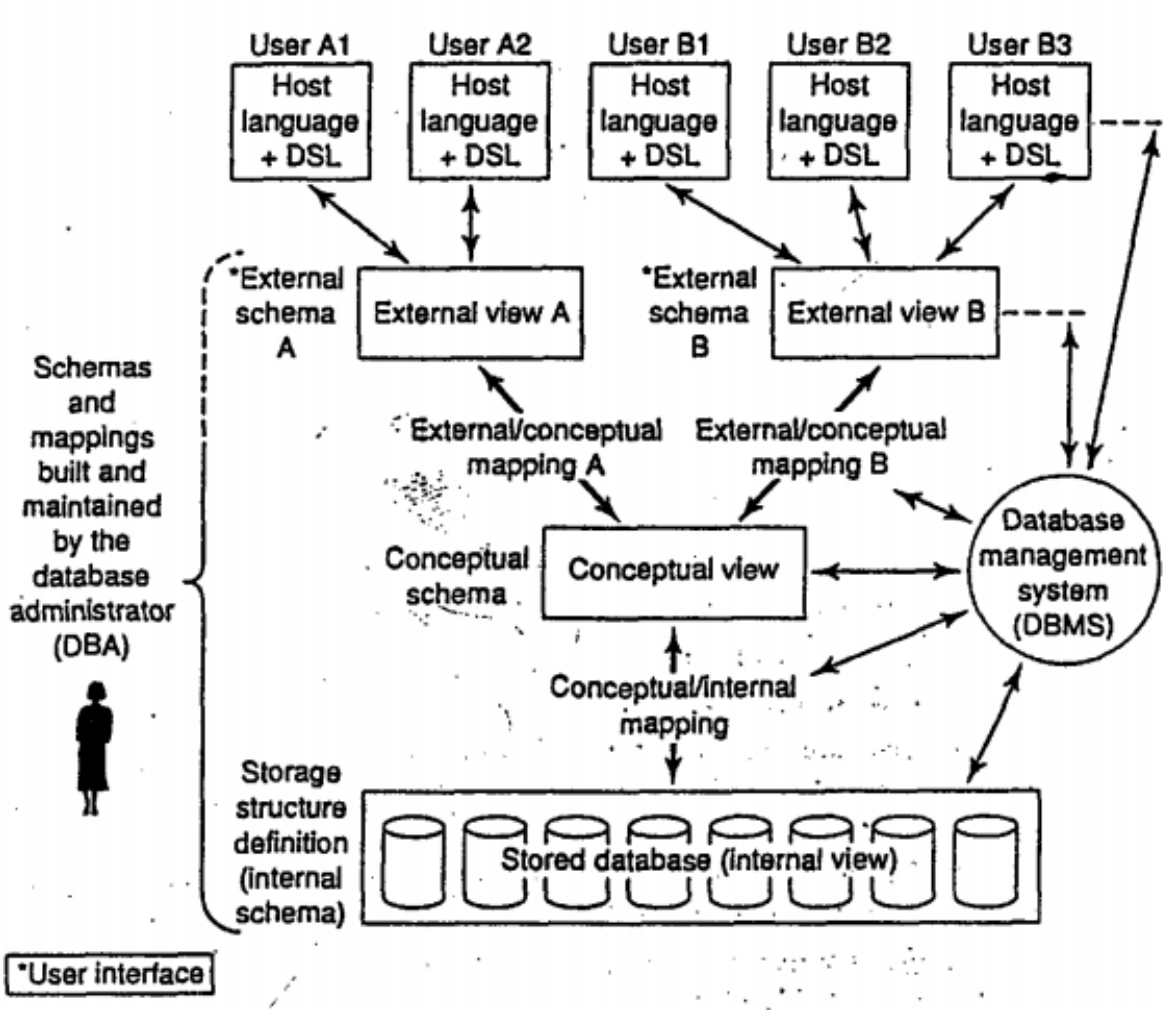
- It allows independent customized user views: Each user should be able to access the same data, but have a different customized view of the data. These should be independent: changes to one view should not affect others.
- It hides the physical storage details from users: Users should not have to deal with physical database storage details.
- The database administrator should be able to change the database storage structures without affecting the users' views.
- The internal structure of the database should be unaffected by changes to the physical aspects of the storage: For example, a changeover to a new disk.

[1] ANSI-SPARC stands for *American National Standards Institute*, Standards Planning And Requirements Committee, *is an abstract design standard for a Database Management System (DBMS)*

ANSI SPARC architecture has three main levels:

- Internal Level
- Conceptual Level
- External Level

- These three levels provide data abstraction, that means they hide the low level complexities from end users .
- Using these three levels, it is possible to use complex structures at internal level for efficient operations andto provide simpler convenient interface at external level.
- The detailed database architecture is shown in the below figure.



The External Level :

- It is the highest level of abstraction, and it describes only part of the entire database.
- This level is the individual user level, closest to the user. It is concerned with the way the data is seen by individual users.
- This level exists to ease the accessibility of the database by an individual user.
- It excludes irrelevant data as well as data which the user is not authorised to access.
- Users can be an application programmer, or an end user of any degree of sophistication. The programmer mayuse either a conventional or proprietary programming language like C, C++, Java, Python. Other users can use either a query language or some special purpose language tailored to that users requirement.
- Even though the conceptual level (logical level) uses simpler structures, complexity remains, as it describes theentire data in the database. The view level describes the portion of the data (stored in the database) relevant to the user.
- The system provides many views for the same database.

- Any database will have only one conceptual and physical schema, but it may have more number of external schemas, each tailored to a particular group of users.
- External Schema is a collection of one or more views and relations from the conceptual schema. By concept aview is a relation, but it is not stored in the DBMS.
- External Schema design is guided by end user requirement

The Conceptual Level:

- The conceptual level is a way of describing what data is stored within the whole database and how the data is inter-related. The conceptual level does not specify how the data is physically stored.
- It is a level of indirection between the other two.
- It is also referred as logical level.
- This level describes data using relatively simple structures. It describes the stored data in terms of the data model of the DBMS.
- DBA uses logical level of abstraction.
- It gives global view of database.
- It is Independent of hardware and software
- This schema defines all the logical constraints that need to be applied on the data stored.
- It defines tables, views, integrity constraints etc.
- The process of arriving at a good conceptual schema is called conceptual database design.

The Internal Level:

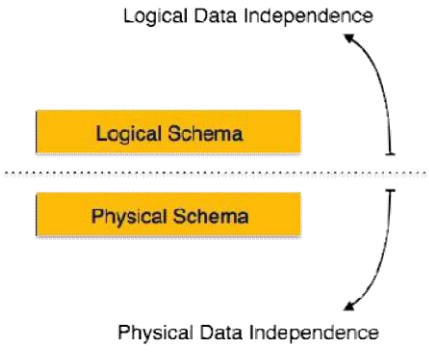
- The lowest level of abstraction describes how the data is actually stored in the memory.
- This level is closest to the physical storage.
- It is also referred as storage level
- It is concerned with the way the data is stored inside the system.
- This level describes complex low-level data structures in detail.
- Here the data structures to store the relations are decided based on the access methods like sequential/random and file organization methods like B+ trees, hashing for sorting etc.
- Decision about the physical schema is based on an understanding of how the data is typically accessed. The process of arriving at a good physical schema is called Physical Database Design.

Advantages of Three Tier DBMS Architecture are:

- **Scalability** - Since the database server isn't aware of any users beyond the application layer and the application layer implements load balancing, there can be as many clients as you want.
- **Data Integrity** - Data corruption and bad requests can be avoided because of the checks performed in the application layer on each client request.
- **Security** - The removal of the direct connection between the client and server systems via abstraction reduces unauthorized access to the database.

Data Independence :

- The main benefit of DBMS is it offers data independence i.e., the application programs are insulated from changes in the way the data is structured and stored and this is achieved through the use of the three levels ofabstraction.
- The main purpose of data abstraction is achieving data independence in order to save time and cost requiredwhen the database is modified or altered.



- We have namely two levels of data independence arising from these levels of abstraction :
    1. Physical data independence
    2. Logical data independence

Physical level data independence :

- Application programs are not affected by changes in physical schema is called physical data independence.
- It refers to the characteristic of being able to modify the physical schema without any alterations to theconceptual or logical schema, done for optimisation purposes,
- e.g., Conceptual structure of the database would not be affected by any change in storage size of the databasesystem server. Changing from sequential to random access files is one such example.
- These alterations or modifications to the physical structure may include:
  - Utilising new storage devices.
  - Modifying data structures used for storage.
  - Altering indexes or using alternative file organisation techniques etc.

Logical level data independence:

- It refers characteristic of being able to modify the logical schema without affecting the external schema orapplication program.
- The user view of the data would not be affected by any changes to the conceptual view of the data.
- These changes may include insertion or deletion of attributes, altering table structures entities or relationshipsto the logical schema etc.

Mappings:

- In addition to the levels, architecture also involves mappings.
  - One conceptual / internal mapping
  - Several external / conceptual mappings

The conceptual / internal mapping:

- It defines the correspondence between the conceptual view and the stored database.
- It specifies how the records and fields at conceptual level are represented in the internal level.
- This mapping has to be changed whenever the structure of database is changed.
- It is the responsibility of the DBA and the underlying DBMS.
- Physical data independence is preserved through this mapping.

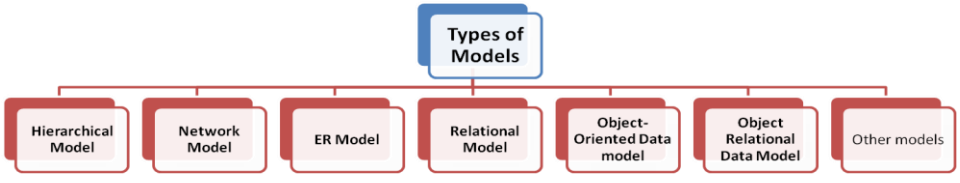
An External / Conceptual Mapping:

- It defines the correspondence between a particular external view and the conceptual view.

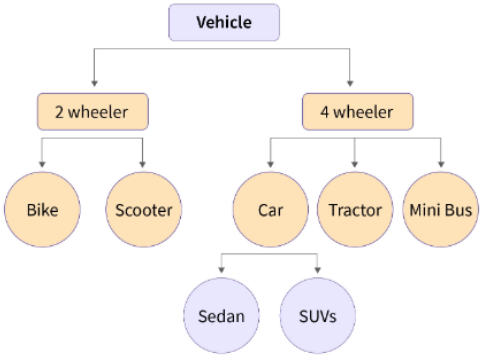
Properties of Database

- Data sharing
- Data Integration
- Data Integrity
- Data Security
- Data Abstraction
- Data Independence

Types of Data Models:



Hierarchical Model : The hierarchical data model is one of the oldest data models, developed in the 1950s by IBM. In this data model, the data is organized in a hierarchical tree-like structure. This data model can be easily visualized because each record has one parent and many children (possibly 0) as shown in the image given below.

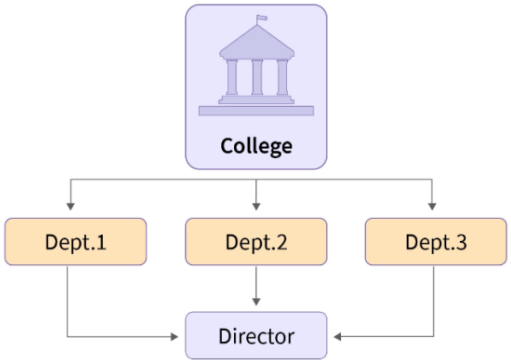


The above given image represents the data model of the Vehicle database, vehicle are classified into two types Viz. two-wheelers and four-wheelers and then they are further classified. The main drawback we can see here is we can only have one to many relationships under this model, hence the hierarchical data model is very rarely used nowadays.

Network Model : A network model is nothing but a **generalization of the hierarchical data model** as this data model allows many to many relationships therefore in this model a record can also have more than one parent.

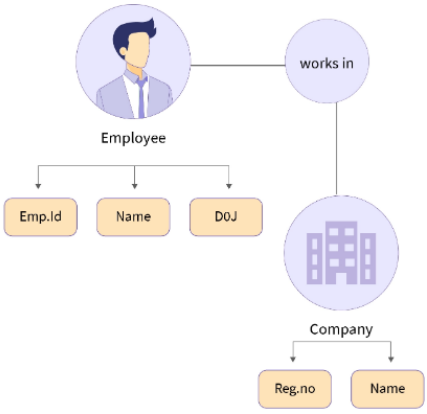
The network model can be represented as a graph and hence it replaces the hierarchical tree with a graph in which object types are the nodes and relationships are the edges.

For example -



Here you can see all the three departments are linked with the director which was not possible in the hierarchical data model. In the network model, there can be many possible paths to reach a node from the root node (College is the root node in the above case), therefore the data can be accessed efficiently when compared to the hierarchical data model. But, on the other hand, the process of insertion and deletion of data is quite complex.

Entity-Relationship model (ER Model) : An Entity-Relationship model is a high-level data model that describes the structure of the database in a pictorial form which is known as ER-diagram. In simple words, an ER diagram are used to represent logical structure of the database easily. ER model develops a conceptual view of the data hence it can be used as a blueprint to implement the database in the future. Developers can easily understand the system just by looking at ER diagram. Let's first have a look at the components of an ER diagram.



- Entity - Anything that has an independent existence about which we collect the data. They are represented as rectangles in the ER diagram. For example - Car, house, employee.
- Entity Set - A set of the same type of entities is known as an entity set. For example - Set of students studying in a college.
- Attributes - Properties that define entities are called attributes. They are represented by an ellipse shape.
- Relationships - A relationship is used to describe the association between entities. They are represented as diamond or rhombus shapes in the ER diagram.

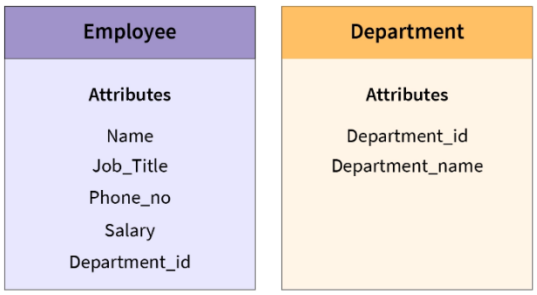
In the above-represented ER diagram, we have two entities that are Employee and Company and the relationship among them. Also, in the above-represented ER diagram, we can see that both employee and company have some attributes and the relationship is of "works in" type, which means employee works in a company.

Relational Model: This is the **most widely accepted data model.** In this model, the database is represented as a **collection of relations in the form of rows and columns of a two-dimensional table.** Each row is known as a tuple (a tuple contains all the data for an individual record) while each column represents an attribute. For example –

Stu. Id	Name	Branch
101	Naman	CSE
102	Saloni	ECE
103	Rishabh	IT
104	Pulkit	ME

The above table shows a relation "STUDENT" with attributes as Stu. Id, Name, and Branch which consists of 4 records or tuples.

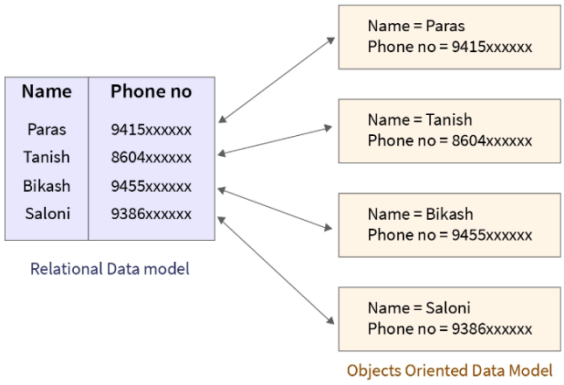
Object-Oriented Data model: As suggested by its name, the object-oriented data model is a combination of object oriented programming, and relational data model. In this data model, the data and their relationship are represented in a single structure which is known as an object. Since data is stored as objects we can easily store **audio, video, images,** etc in the database which was very difficult and inconvenient to do in the relational model. As shown in the image below two objects are connected with each other through links.



In the image, we have two objects that are Employee and Department in which all the data is contained in a single unit (object). They are linked with each other as they share a common attribute i.e.i.e. Department_Id.

Object Relational Data Model : Again as suggested by its name, the object-relational data model is an integration of the object oriented model and the relational model. Since it inherits properties from both of the models it supports objects, classes, etc like object oriented model and tabular structures like the relational model. For example -

It provides data structures and operations used in the relational model and also provides features of object oriented models like classes, inheritance, etc. The only drawback of this data model is that it is complex and quite difficult to handle.



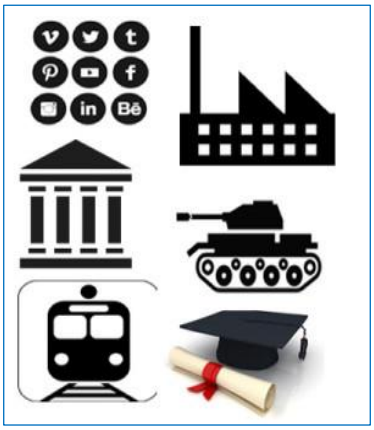
Advantages of Data Models

- Data models ensure that the data is represented accurately.
- The relationship between the data is well defined.
- Data redundancy can be minimized and missing data can be identified easily.
- Last but not the least, security of the data is not compromised.

Disadvantages of Data Models

- The biggest disadvantage of the data model is, one must know the characteristics of physical data to build a data model.
- Sometimes in big databases, it is quite difficult to understand the data model also the cost incurred is very high.

Applications of DBMS



Database is widely used. The some of the representative applications are:

1. **Banking:** for customer information, accounts and loans and banking transactions.
2. **Universities:** for student registrations and grades.
3. **Online shopping:** Everyone wants to shop from home. Everyday new products are added and sold only with the help of DBMS. Purchase information, invoice bills and payment, all of these are done with the help of DBMS.
4. **Airlines:** for reservations and schedule information.
5. **Credit card transactions :** for purchases on credit cards and generation of monthly statements.
6. **Library Management System:** maintain all the information relate to book issue dates, name of the book,author and availability of the book.
7. **Telecommunications:** for keeping records of call made, generating monthly bills, maintaining balances onprepaid calling cards.
8. **Sales:** for customer, product and purchase information.
9. **Finance:** for storing information about holdings, sales, and purchases of financial instruments such as stocks and bonds.
10. **Manufacturing:** for management of supply chain and for tracking production of items in factories, inventoriesof items and orders for items.
11. **Human Resource:** for information about employees, salaries, payroll taxes and benefits.

Advantages of DBMS

1. **Controlling of Redundancy:** Data redundancy refers to the duplication of data (i.e storing same data multiple times). In a database system, by having a centralized database and centralized control of data by the DBA the unnecessary duplication of data is avoided. It also eliminates the extra time for processing the large volume of data. It results in saving the storage space.
2. **Improved Data Sharing:** DBMS allows a user to share the data in any number of application programs.
3. **Data Integrity:** Integrity means that the data in the database is accurate. Centralized control of the data helps in permitting the administrator to define integrity constraints to the data in the database. For example: in customer database we can enforce an integrity that it must accept the customer only from Noida and Meerut city.
4. **Security:** Having complete authority over the operational data, enables the DBA in ensuring that the only mean of access to the database is through proper channels. The DBA can define authorization checks to be carried out whenever access to sensitive data is attempted.
5. **Data Consistency:** By eliminating data redundancy, we greatly reduce the opportunities for inconsistency. For example: is a customer address is stored only once, we cannot have disagreement on the stored values. Also updating data values is greatly simplified when each value is stored in one place only. Finally, we avoid the wasted storage that results from redundant data storage.
6. **Efficient Data Access:** In a database system, the data is managed by the DBMS and all access to the data is through the DBMS providing a key to effective data processing
7. **Enforcements of Standards:** With the centralized of data, DBA can establish and enforce the data standards which may include the naming conventions, data quality standards etc.
8. **Data Independence:** Ina database system, the database management system provides the interface between the application programs and the data. When changes are made to the data representation, the meta data obtained by the DBMS is changed but the DBMS is continues to provide the data to application program in the previously used way. The DBMs handles the task of transformation of data wherever necessary.
9. **Reduced Application Development and Maintenance Time :** DBMS supports many important functions that are common to many applications, accessing data stored in the DBMS, which facilitates the quick development of application.

Disadvantages of DBMS

1. Increased Complexity
2. Requirement of New and Specialized Manpower
3. Large Size of DBMS

Important Questions

1. What is the need of data model in DBMS and give its classification
2. Who are the different database users? Explain their interfaces to database management system.
3. Describe the client server architecture for the database with necessary diagram.
4. Define Database Management Systems.
5. What is Data Base Administrator? Discuss the functions of DBA.
6. Explain DBMS applications
7. What are the disadvantages in file system?
8. What is data independence? Discuss three tire schema architecture of data independence.
9. Explain storage manager component.
10. Explain object-oriented data model.
11. Explain briefly the languages supported by database systems.
12. What is Data modeling? Explain relational model.
13. List various types of database users. Explain.
14. Discuss abstract view of data with diagram.
15. Explain about Entity-Relationship model with an example.
16. Define the two levels of data independence.
17. Explain the merits and demerits of data base system.
18. Differentiate between schema and instance.
19. Describe the characteristics of a database system.
20. Draw and explain three-tier schema architecture of database system.
21. Present any two database applications by describing their features.
22. What do you mean by environment in database systems? Explain with the help of database system structures.
23. Mention various groups of database users. Explain about their roles in detail.
24. What is a data model? Describe various data models.
25. Distinguish between centralized and client-server architectures of a database system.
26. Differentiate between File system and Database System
27. List out Database applications.
28. Explain in detail about Database Management System advantages over file management system.
29. Explain the concept of Data independence.
30. Briefly describe various architectures of database systems.
31. What is Data Independence? Why is it essential?

Relation Algebra and Calculus

- These are formal query languages associated with the relational model.
- Query Languages are specialized languages for asking questions / queries that involve data in the database.
- Queries in Relational algebra (RA) are composed using a collection of operators. Each query describes a step bystep procedure for computing the desired answer.
- Query in Relational Calculus describes the desired answer without specifying how the answer is computed. This is non-procedural and this style of querying is called declarative.
- These formal query languages greatly influenced commercial query languages such as SQL.

Sample Queries are presented using the following Schema

Sailors (sid : integer, sname : string, rating : integer)
Boats (bid : integer, bname : string, color : string)
Reserves (sid : integer, bid : integer, day : date)

Instance S1 of Sailors

sid	sname	rating	age
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Instance S2 of Sailors

sid	sname	rating	age
28	Yuppy	9	35.0
31	Lubber	8	55.5
44	Guppy	5	35.0
58	Rusty	10	35.0

Instance R1 of Reserves

E	bid	day
22	101	10/10/96
58	103	11/12/96

Relational Calculus:

- It is an alternative to Relational Algebra.
- It allows us to describe the set of answers without being explicit about how they should be computed. i.e., it isa non-procedural language. It can also be called as declarative language.
- Calculus has variables, constants, comparison operators, logical connectives and quantifiers
- Relational calculus comes in two flavours
  - Tuple Relational Calculus (TRC)
  - Domain Relational Calculus (DRC)
- Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called formulas. An answer tuple is essentially an assignment of constants tovariables that make the formula evaluate to true.

Tuple Relational Calculus:

- In TRC, variables range over tuples. i..e, variables in TRC take on tuples as values.
- A tuple variable is a variable that takes on tuples of a particular relation schema as values.
- It has more influence on SQL.
- A TRC query has the form

{ T | P(T) }

Where T is a tuple variable.
P(T) denotes a formula that describes T.

- Result of { T | P(T) } this query is the set of all tuples *t* which the formula P(T) evaluates to true.

Ex: Find all sailors with a rating above 7.

$$\{S \mid S \in \text{Sailors} \wedge s.\text{rating} > 7\}$$

Syntax of TRC queries:

Let *Rel* be a relation name,*R* and *S* be tuple variables,
a be an attribute of R, and *b* be an attribute of S,
let *op* denotes a relational operator.
An atomic formula is one of the following:

- $R \in \text{Rel}$
- $R.a\ \text{op}\ S.b$
- $R.a\ \text{op}\ \text{Constant}$

A formula is recursively defined to be one of the following, where P and Q are themselves formulas and P(R) denotes aformula in which the variable R appears.

- any atomic formula
- $\neg p$, $P \wedge q$, $P \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

- The quantifiers are said to bind the variable.
- A variable is said to be free in a formula if the formula does not contain an occurrence of a quantifier

that binds it.

- A TRC query is defined to be an expression of the form {T | P(T)}, where T is the only free variable in theformula P.

Examples :

1. **Find the names and ages of sailors with a rating above 7.**

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

This query illustrates a useful convention: P is considered to be a tuple variable with exactly two fields, which are called name and age, because these are the only fields of P that are mentioned and P does not range over any of the relations in the query; that is, there is no subformula of the form P ∈ Relname. The result of this query is a relation with two fields, name and age. The atomic formulas P.name = S.sname and P.age = S.age give values to the fields of an answer tuple P . On instances B1, R2, and S3, the answer is the set of tuples ⟨Lubber, 55.5⟩, ⟨Andy, 25.5⟩, ⟨Rusty, 35.0⟩, ⟨Zorba, 16.0⟩, and ⟨Horatio,35.0⟩.

2. **Find the sailor name, boat id and reservation date for each reservation.**

$$\{P \mid \exists R \in Reserves\ \exists S \in Sailors\ (R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

For each Reserves tuple, we look for a tuple in Sailors with the same sid. Given a pair of such tuples, we construct an answer tuple P with fields sname, bid, and day bycopying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple.

3. **Find the names of sailors who have reserved boat 103.**

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the sid field, andwith bid = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, sname

4. **Find the names of sailors who have reserved a red boat.**

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that S.sid = R.sid, R.bid = B.bid, and B.color = 'red'.

5. **Find the names of sailors who have reserved at least two boats.**

$$\{P \mid \exists S \in Sailors\ \exists R1 \in Reserves\ \exists R2 \in Reserves\ (S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

6. Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors\ \forall B \in Boats\ (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Notice how easily it is expressed in the calculus. The calculus query directly reflects how we might express the query in English: "Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B."

7. Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \in \forall B \in Boats(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply the sailor having reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor's name, we have avoided introducing a new free variable.

We can write this query without using implication, by observing that an expression of the form p⇒ q is logically equivalent to ¬p ^ q:

$$\{S \mid S \in Sailors \in \forall B \in Boats\ (B.color \neq 'red' \cup (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: "Find sailors S such that for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B."

Domain Relational Calculus:

- A domain variable is a variable that ranges over the values in the domain of some attribute.
- Form of a DRC query.

  $\{\langle x_1, x_2, \ldots, x_n\rangle \mid P(\langle x_1, x_2, \ldots, x_n\rangle)\}$

  Each x_i is a domain variable.

  $P(\langle x_1, x_2, \ldots, x_n\rangle)$ denotes a DRC formula.

- The result of this query is the set of all tuples $\langle x_1, x_2, \ldots, x_n\rangle$ for which the formula evaluates to true.

  Answer includes all tuples $\langle x1, x2, \ldots, xn\rangle$ that make the *formula* $p(\langle x1, x2, \ldots, xn\rangle)$ be *true*.

- DRC formula is similar to TRC formula DRC formula:

  Atomic formula:
  - $\langle x1, x2, \ldots, xn\rangle \in Rname$, or X *op* Y, or X *op* constant
  - *op* is one of $<, >, =, \leq, \geq, \neq$

  Formula:
  - an atomic formula, or
  - $\neg p, p \wedge q, p \vee q$, where p and q are formulas, or
  - $\exists X(p(X))$, where variable X is *free* in p(X), or
  - $\forall X(p(X))$, where variable X is *free* in p(X)

  The use of quantifiers $\exists X$ and $\forall X$ is said to *bind* X.
  - A variable that is not bound is free.

Examples :

1. **Find the sailors with a rating above 7.**

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge T > 7\}$$

2. **Find the names of sailors who have reserved boat 103.**

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \exists\langle Ir,Br,D\rangle \in Reserves(Ir = I \wedge Br = 103))\}$$

(or)

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \exists D(\langle I,103,D\rangle \in Reserves))\}$$

3. **Find the names of sailors who have reserved at least two boats.**

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Br1, Br2, D1, D2(I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 = Br2)$$

4. **Find the names of sailors who have reserved all boats.**

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \forall B,BN,C(\neg(\langle B,BN,C\rangle \in Boats) \vee (\exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

5. **Find sailors who have reserved all red boats.**

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats (C = 'red' \Rightarrow \exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

Database Design

Database Design process is divided into 6 basic steps.

1. Requirements Analysis
2. Conceptual Database Design
3. Logical Database Design
4. Schema Refinement
5. Physical Database Design
6. Application and Security Design.

1. Requirements Analysis

- This is the first step in designing any database application.
- This is an informal process that involves
  - Discussion with users,
  - Study of current environment and its changes
  - Analysis of documentation on existing applications.

- Under this, we have to understand the following.
  - What data is to be stored in a database?
  - What applications must be built on top of the database?
  - What users want from the database?

- Example: For customer database, data is cust-*name, cust-city, and cust-no.*

- There are several methods today to organize and present the information gathered in requirements analysis.

2. Conceptual Database Design

- The information gathered in the requirements analysis step is used to develop a higher-level description andconstraints of the data.
- This phase is carried out by ER model.
- The goal here is to create a simple description of data that closely matches how users and developers think ofdata.
- Characteristics of this phase are as below.
  1. ***Expressiveness***: The data model should be expressive to distinguish different types of data, relationships and constraints.
  2. ***Simplicity and Understandability :*** The model should be simple to understand the concepts.
  3. ***Minimality:*** The model should have small number of basic concepts.
  4. ***Diagrammatic Representation:*** The model should have a diagrammatic notation for displaying the conceptual schema.
  5. ***Formality:*** A conceptual schema expressed in the data model must represent a formal specification of thedata.
- Example: *Cust_name : string;*
  Cust_no : integer;
  Cust_city : string;

3. Logical Database Design :

- Under this, we must choose a DBMS to implement our database design and convert the conceptual databasedesign into a database schema.
- The choice of DBMS is governed by number of factors as below.
  1. Economic Factors.
  2. Organizational Factors.
- As we are considering relational DBMS, here we convert ER diagram into a relational database schema.

4. Schema Refinement :

- Under this, we have to analyze the collection of relations in our relational database schema to identify thepotential problems.

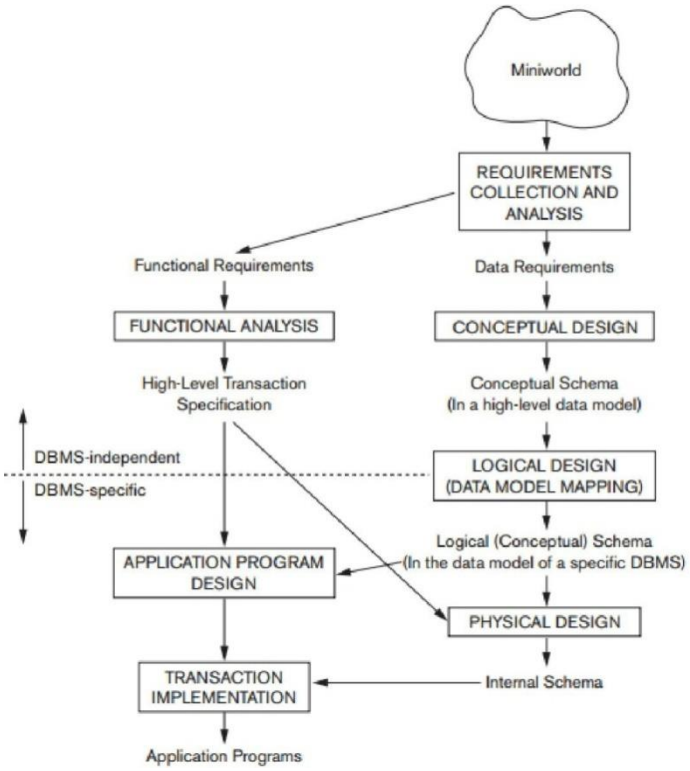
5. Physical Database Design

- Physical database design is the process of choosing specific storage structures and access paths forthe database files to achieve good performance for the various database applications.
- This step involves building indexes on some tables and clustering some tables.
- The physical database design can have the following options.
  1. ***Response Time***: This is the elapsed time between submitting a database transaction for execution andreceiving a response.
  2. ***Space Utilization:*** This is the amount of storage space used by the database files and their access pathstructures on disk including indexes and other access paths.
  3. ***Transaction Throughput:*** This is the average number of transactions that can be processed per minute.

6. Security Design:

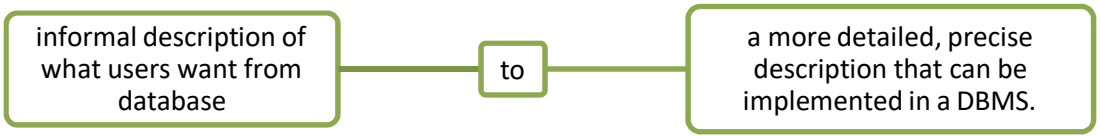
- In this step, we must identify different user groups and different roles played by various users.
- For each role, and user group, we must identify the parts of the database that they must be able to access

A Simple diagram to show the main phases of database design



The Entity Relationship Model

- Data Model allows us to describe the data involved in a real world enterprise in terms of objects and theirrelationships.
- ER Model is widely used to develop initial database design.
- It allows us to move from



Entities, Attributes and Entity sets

The ER model describes data as entities, relationships, and attributes.

Entity :

- An entity is a thing or object in the real world with an independent existence.
- An entity may be an object
  - with a *physical* existence (for example, a particular person, car, house, or employee) or
  - it may be an object with a *conceptual* existence (for instance, a company, a job, or a university course).
- Collection of similar entities is referred as Entity Set
  - Eg: Faculty set
- Entity Sets need not be disjoint.
  - Eg: A student who is part of CSE entity set may also be a part of CRT entity set.
- Entities are represented by Rectangle with entity name inside.

Attribute :

- Attributes are particular properties that describe the entity.
- All entities in a given entity set have the same attributes.
  - Eg: Attributes of Employee entity set are empno, name, salary .
- It is represented by an oval.

- The attribute value that describes it becomes major part of the data stored in the Database.
- **Domain** is set of possible values for an attribute. For every attribute of an entity set, we must identify adomain.
  - Eg: employee name is set of 50 character string.
- **NULL values :** sometimes a particular entity may not have an applicable value for an attribute or the value is unknown. To handle these situations a special value NULL is used.
  - Eg: Flat number in the address, college degree of a person(may or may not exist)

Several types of attributes occur in the ER model:

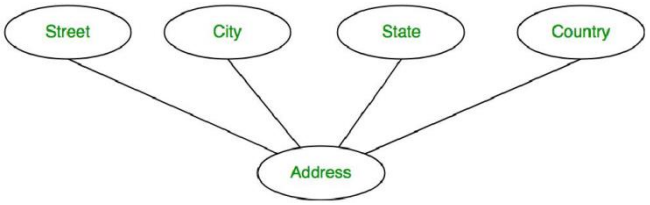
1. Simple Attributes
2. Composite Attributes
3. Singlevalued Attributes
4. Multivalued Attributes
5. Stored Attributes
6. Derived Attributes
7. Key Attributes
8. Complex Attribute

Simple Attribute :

- The attributes which cannot be further subdivided.
  - Eg. Age, marital status, city etc.

Composite Attributes :

- The attributes which can be further divided into more attributes with independent meanings.
  - Eg: Address can be further divided into street, city, state, country.
- The notation used is :



Single Valued Attribute:

- Attribute which has single value for a particular entity is single-valued attribute.
  - Eg: age of a person.

Multivalued Attribute :

- Attributes which can have multiple values for a single entity.
  - Eg: color of a car, phone number.
- These attributes are represented using a double lined oval.

Stored Attribute :

- Attributes that cannot be derived from other attributes.
  - Eg: Birth_date

Derived Attribute:

- Attributes which are derived from other attributes.
- This can be derived from one or more attributes or from a separate table.
  - Eg: age of a person (can be derived from DOB), experience of an employee, commission.
- These are represented using a dotted oval.

Complex Attributes:

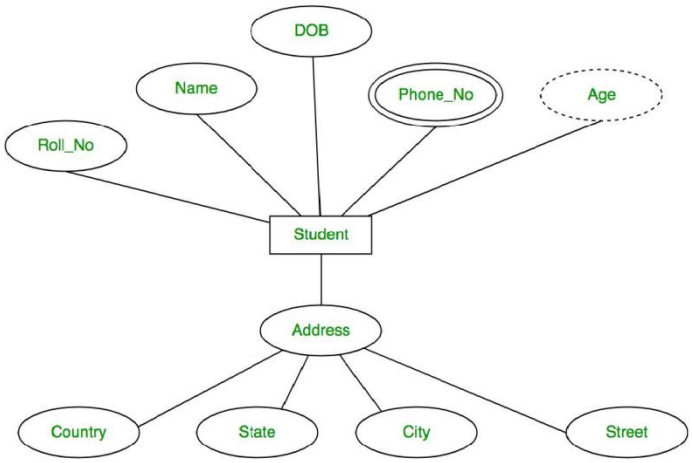
- An attribute which is made by using the multivalued attributes and composite attributes.
  - Eg: A person can have more than one residence; each residence can have more than one phone.

Key Attributes :

- The attribute which **uniquely identifies each entity** in the entity set is called key attribute.
  - For example, Roll_No will be unique for each student.
- These attributes have unique values for all entities.
- In ER diagram, key attribute is represented by an oval with underlying line.

- Some entity sets have more than one key.

The complete entity type **Student** with its attributes can be represented as:



Relationships and Relationshipsets

- A **relationship** is an association among one or more entities.
  - *Eg: Karthika works in CSE department.*
    Karthika is an entity of Employee entity set, CSE is an entity of department entity set and works in is the relation between these two entities.
- A **relationship set** is a collection of similar relationships
- It is a set of n tuples

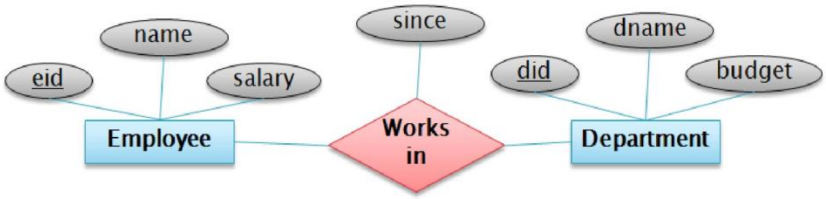
$$\{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$

- Each tuple denotes a relationship involving N entities e_1 through e_n, where entity e_i is in the entity set E_i.
- An entity set can participate in several relationship sets.

Descriptive attributes: A relationship can have attributes which describes about the relation. These are usedto record information about the relationship, rather than about any one of the participating entities.

- Eg: Latha works in CSE department since 01/01/15.

Eg: The works in relationship



- The relationships are identified the participating entities, not by descriptive attributes.
  - i.e., in the above example relationship is identified by eid and did.

- An **instance** of a relationship set is a set of relationships. It can be thought of as a snapshot of relationship set at some instant in time.

Degree of a relationship :

The number of different entity sets participating in the relationship set is the degree. It can be

- Unary relationship
- Binary relationship
- Ternary relationship
- N-ary relationship

Unary relationship :

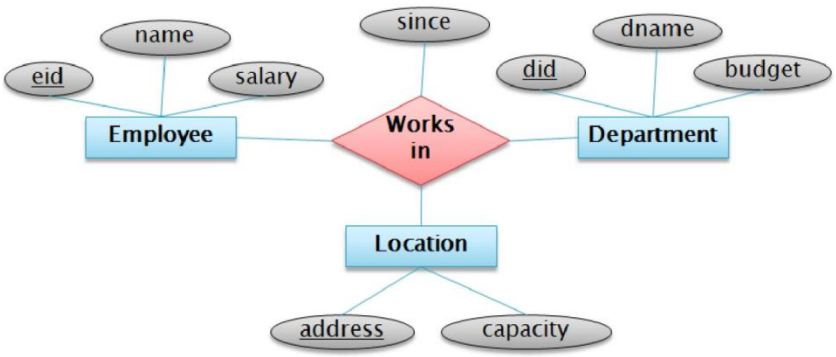
- This is a relationship between the entities of the same entity set.
- The entity sets that participate in a relationship set need not be distinct.
  - Eg*: Latha reports to Sumit.*
    Both Latha and Sumit are entities of Employee Entity set, but they are playing different roles. Here they are supervisor and subordinate.

Binary relationship :

- This is a relationship between any two different entities.
  - Eg: The worksin relationship shown above is a binary relationship. Because the participating entitiesare only two (Employee and Department)

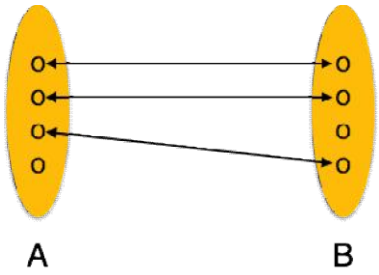
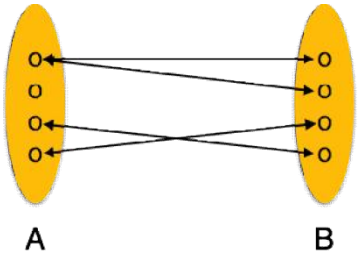
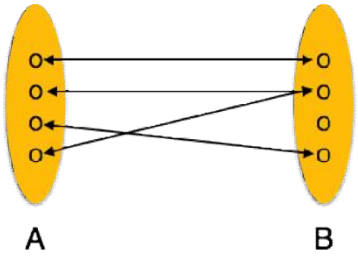
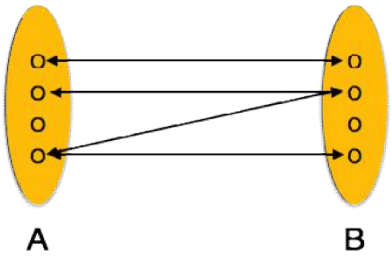
Ternary Relationship:

- This is a relationship between any three different entities.
  - Eg: Each department has offices in several locations. The ER diagram to record the locations at which each employee works is



Cardinality of a relationship:

- The number of times an entity of an entity set participates in a relationship.
- Various cardinality ratios are

one – to – one	One entity from entity set A can be associated with at most one entity of entity set Band vice versa. It is marked as **1 : 1** 
one – to – many	One entity from entity set A can be associated with more than one entities of entity set B however an entity from entity set B, can be associated with at most one entity. It is marked as **1 : N** 
many – to – one	More than one entities from entity set A can be associated with at most one entity of entity set B, however an entity from entity set B can be associated with more than one entity from entity set A. It is marked as **N : 1** 
many – to – many	One entity from A can be associated with more than one entity from B and vice versa. It is marked as **M : N** 

Participation Constraint:

- It specifies whether the existence of an entity depends on its being related to another entity via a relationshiptype.
- This constraint specifies the minimum number of relationship instances that each entity can participate in.
- There are two types of participation constraints

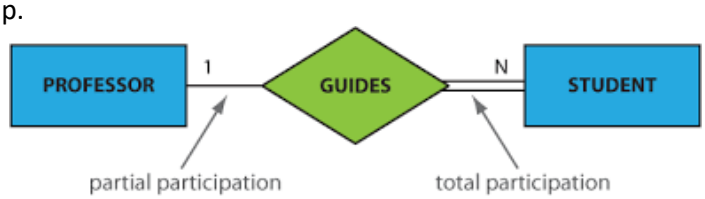
1. Total participation
2. Partial participation

- **Total Participation :** Every entity in the entity set must be related to another entity in the relationship set.
  - Eg: Every employee must work for a department.

In ER diagrams the total participation is represented by double line between participating entity type torelationship.

- **Partial Participation :** Some part of the entityset is related to some other entity via a relationship.
  - Eg: Few of the employee are managing the department.

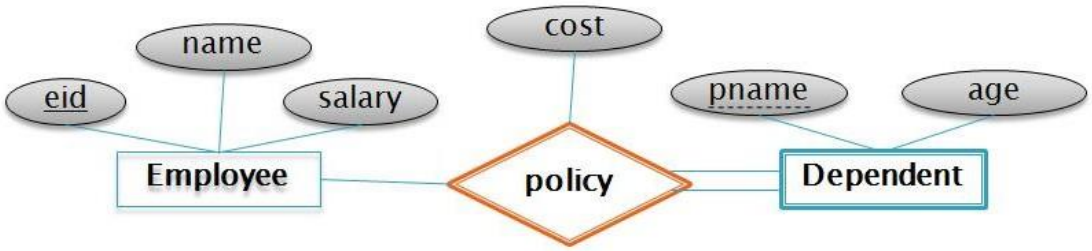
In ER diagrams the total participation is represented by solid line between participating entity type torelationship.



  - Consider the relationship - Employee is head of the department. Here all employees will not be the head of the department. Only one employee will be the head of the department. In other words, only few instances of employee entity participate in the above relationship. So employee entity's participation is partial in the said relationship. However each department will be headed by some employee. So department entity's participation n is total in the said relationship.

Weak Entity Types :

- **Weak Entity** : Entity type that do not have key attributes of their own is called a Weak Entity.
- **Strong / Regular Entity :** Entity types that do have a key attribute.
- Entities of a weak entity type are identified by another entity types attribute value. That another entity type isidentifying or owner identity type.
  - Eg: employee



- As weak entities cannot be identified on their own, they always has a total participation w.r.t to its identifyingrelationship.

- **Partial Key :** the attribute that can uniquely identify weak entities that are related to the same owner entity.
  - Eg: pname (name of the dependent)

- In ER diagram, the weak entity types are represented by double lined rectangle.

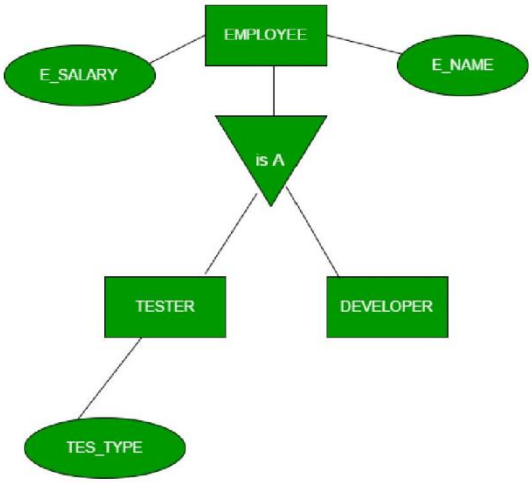
- And identifying relationship is represented by double lined diamond.

- Partial key is underlined with a dashed or dotted line.

EER Model

- It is natural to classify the entities in an entity set into subclasses.
- EER model stands for Enhanced Entity Relationship Model.
- It includes all the modelling concepts of ER model and also the concepts of subclass and superclass and the related concepts of specialization and generalization. The resultant diagrams are called Enhanced ER diagrams.

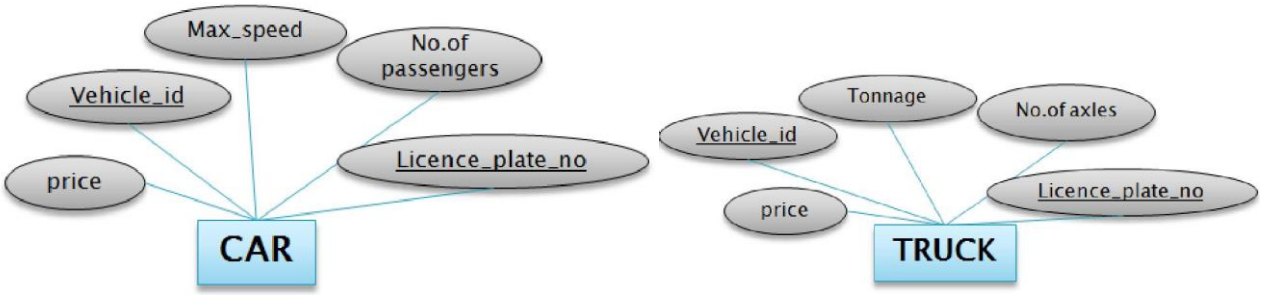
- **Specialization:**
  - Specialization is the process of identifying subsets of an entity set that share some distinguishing characteristic.
  - The entity type from which subclasses are defined is called superclass of specialization.
  - Eg:
    - set of classes Secretary, Technician, engineer are subclasses of superclass employee. This distinguishing is on the basis of job type.
    - Another specialization on employee based on method_of_pay can be hourly_employee and salaried_employee.
- **we may have several specializations of the same entity type based on different distinguishing characteristic**
- Attributes of subclass are called specific attributes.



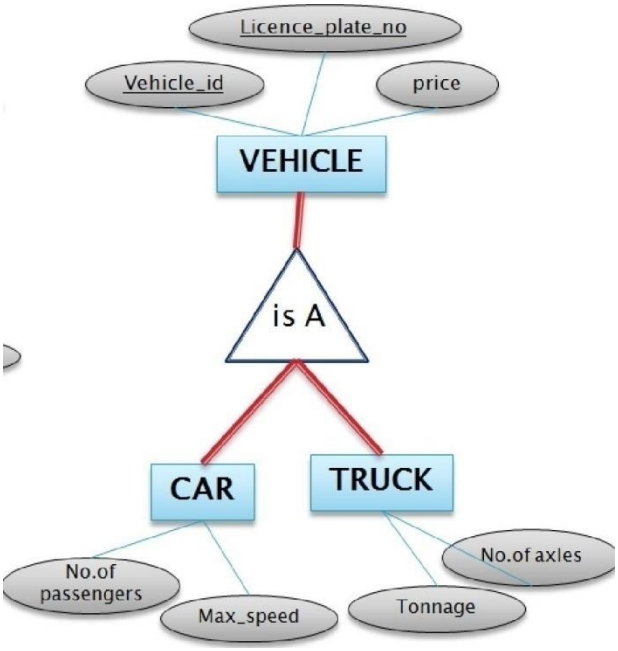
- In the above ER diagram, two specialized entities were identified based on job type.

- Main reasons for including class/subclass relationships and specializations
  1. Certain attributes may apply to some but not all entities of the superclass entity type. A subclass is defined inorder to group the entities to which these attributes apply.
  2. Some relationship types may be participated in only by entities that are members of the subclass.
  3. We can add descriptive attributes that make sense only for the enetities in a subclass.

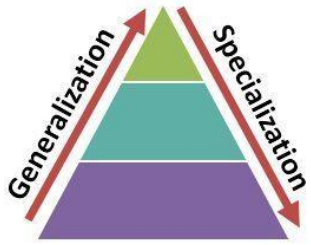
- **Generalization:**
  - This is reverse process of abstraction.
  - Generalization is the process of defining generalized entity type from the given entity types.
  - Here we suppress the differences among several entity types, identify their common features, andgeneralize them into a single super class.
  - Eg: consider two entity types CAR and TRUCK



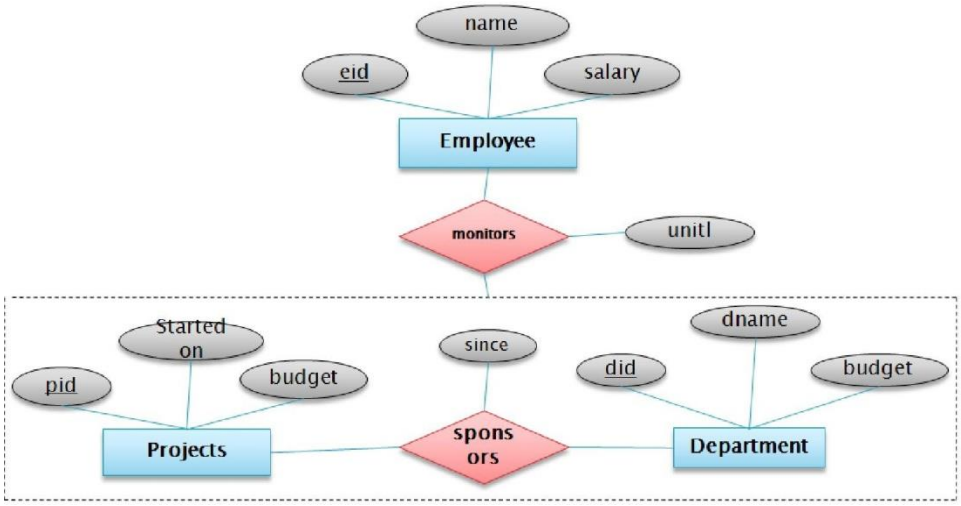
- From the above two entites a generalized entity can identified with the attributes vehicle_id, licence_plate_no, price.



- Generalization is Bottom-Up approach and Specialization is Top-down approach.



- **Aggregation**:
  - A relationship is an association between entitysets. But sometimes Database Designers may need to model a relationship between a collection of entities and relationships.
  - Aggregation is a feature, which allows us to indicate that a relationship participates in another relationship.
  - Eg: Each project entity is sponsored by one or more departments. Sponsors relationship captures this information. The department that sponsors a project might assign employees to monitor the sponsorship. Monitors is a relationship that is associated with sponsors relationship.
  - We use aggregation, to express relationship among relationships.

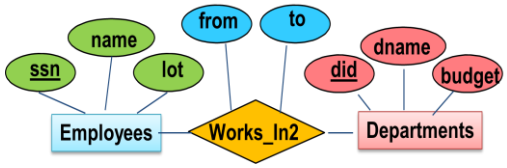


Conceptual Design with ER Model

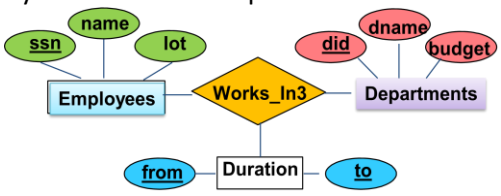
- ER modeling can get tricky!
- Design choices:
  - Should a concept be modeled as an entity or an attribute?
  - Should a concept be modeled as an entity or a relationship?
  - Identifying relationships: Binary or ternary? Aggregation?
- Note constraints of the ER Model:
  - A lot of data semantics can (and should) be captured.
  - But some constraints cannot be captured in ER diagrams.
    - We'll refine things in our logical (relational) design

Entity vs. Attribute:

- ✓ Should address be an attribute of Employees or an entity (connected to Employees by a relationship)?
- ✓ Depends upon the use we want to make of address information, and the semantics of the data:
  - If we have several addresses per employee, address must be an entity (since attributes cannot be set-valued).
  - If the structure (city, street, etc.) is important, e.g., we want to retrieve employees in a given city, address must be modeled as an entity (since attribute values are atomic).



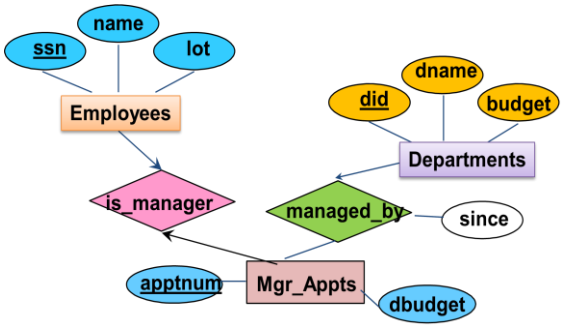
- ✓ Works_In2 does not allow an employee to work in a department for two or more periods.



- ✓ Similar to the problem of wanting to record several addresses for an employee: we want to record several values of the descriptive attributes for each instance of this relationship.

Entity versus Relationship: There is at most one employee managing a department, but a given employee could manage several departments; we store the starting date and discretionary budget for each manager-department pair. This approach is natural if we assume that a manager receives a separate discretionary budget for each department that he or she manages.

What if manager's dbudget covers all managed depts? (can repeat value, but such redundancy is problematic)



Advantages of ER Modeling

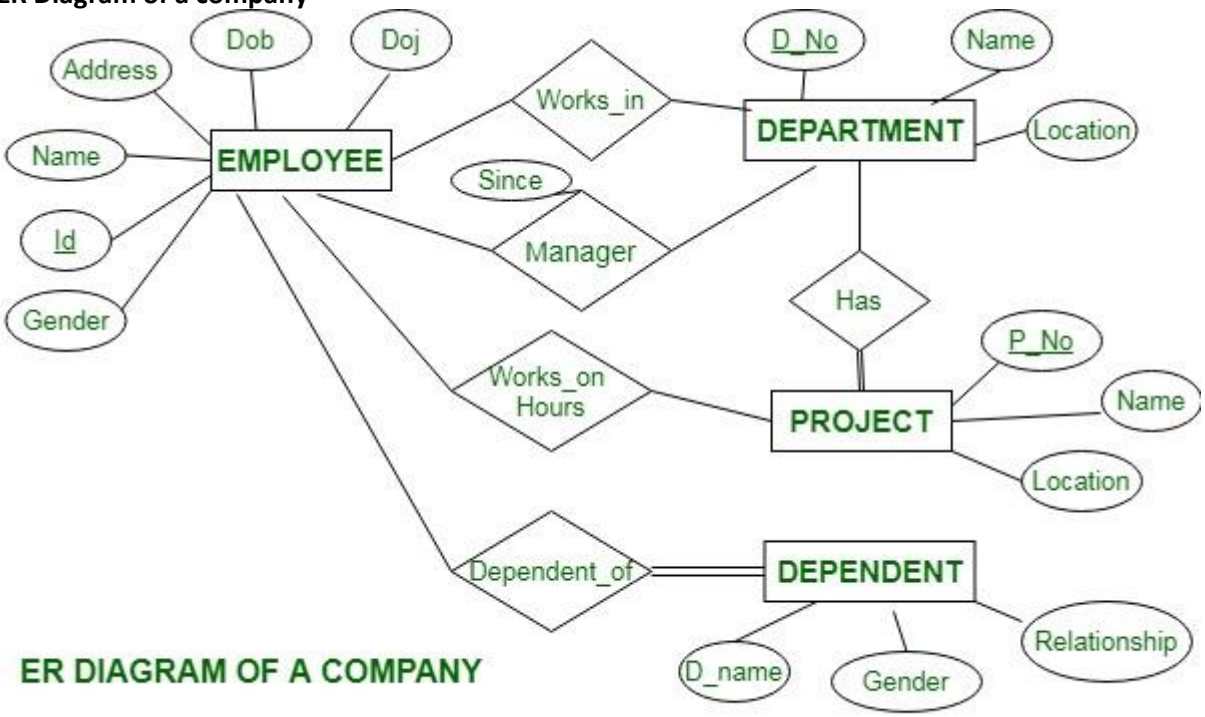
1. ER Modeling is simple and easily understandable. It is represented in business users language and it can be understood by non-technical specialist.
2. Intuitive and helps in Physical Database creation.
3. Can be generalized and specialized based on needs.
4. Can help in database design.
5. Gives a higher level description of the system.

Disadvantages of ER Modeling

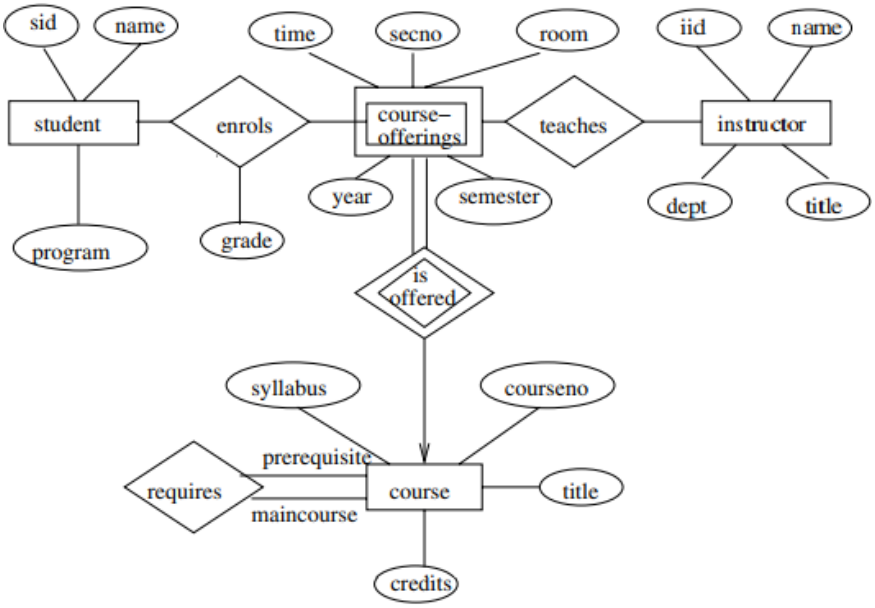
1. Physical design derived from E-R Model may have some amount of ambiguities or inconsistency.
2. Sometime diagrams may lead to misinterpretations

Example:

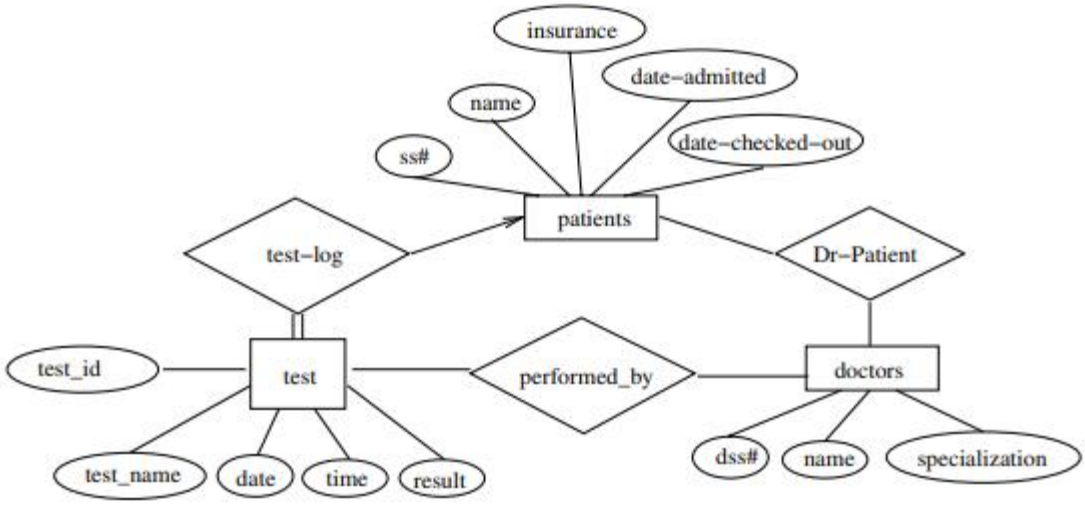
- **ER Diagram of a company**



- ER-Diagram on University



- ER-Diagram on Hospital



Important Questions

1. How to represent a weak entity set in ER diagram? Quote suitable example.
2. Define the following terms and give examples
   a) cardinality (ii)unary relationships (iii)aggregation (iv)specialization
3. How to maintain class hierarchies in ER-Diagrams? Explain with employee database.
4. Explain the following terms:
   a) Entity and entity set.
   b) Attribute and attribute sets.
   c) Relationship and relationship sets.
5. Define generalization and aggregation. Demonstrate generalization and aggregation using E-R diagram.
6. Explain about Entity-Relationship model with an example.
7. Explain about domain constraints and key constraints.
8. What are the major components used in E-R diagram design?
9. Differentiate between super class and sub class.
10. Construct an ER diagram for university registrar's office. The office maintains data about each class, including the instructor, the enrollment and the time and place of the class meetings. For each student class pair a grade is recorded.
11. Determine the entities and relationships.

12. Explain the following:
    a) Ternary relationship b) Weak entity set c) Grouping d) Aggregation.
13. What is ER model? Explain its concepts.
14. Describe entities and relationships with examples.
15. Write about different types of attributes in ER model. Show the notation of each.
16. What is a weak entity type? How to model it? Explain with suitable example.
17. Explain in detail about inheritance, specialization and generalization using ER diagrams.
18. Differentiate specialization and generalization.
19. With the aid of appropriate examples, describe how to model the following in ER model:
    a) Entity type ii) Relationship type iii) Super class iv) Sub class
20. Draw an ER diagram for Hospital management system.
21. Explain about various constraints used in ER-model.
22. Explain the difference among Entity, Entity Type & Entity Set
23. Specify and explain various structural constraints of relationship type.