SQL

1. **SQL**: Structured Query Language, used to access and manipulate data.
2. SQL used **CRUD** operations to communicate with DB.
    1. **CREATE** - execute INSERT statements to insert new tuple into the relation.
    2. **READ** - Read data already in the relations.
    3. **UPDATE** - Modify already inserted data in the relation.
    4. **DELETE** - Delete specific data point/tuple/row or multiple rows.
3. **SQL is not DB, is a query language.**
4. What is **RDBMS**? (Relational Database Management System)
    1. Software that enable us to implement designed relational model.
    2. e.g., MySQL, MS SQL, Oracle, IBM etc.
    3. Table/Relation is the simplest form of data storage object in R-DB.
    4. **MySQL** is open-source RDBMS, and it uses SQL for all CRUD operations
5. **MySQL** used client-server model, where client is CLI or frontend that used services provided by MySQL server.
6. **Difference between SQL and MySQL**
    1. SQL is Structured Query language used to perform CRUD operations in R-DB, while MySQL is a RDBMS used to store, manage and administrate DB (provided by itself) using SQL.

SQL DATA TYPES (Ref: hUps://www.w3schools.com/sql/sql_datatypes.asp)

1. In SQL DB, data is stored in the form of tables.
2. Data can be of different types, like INT, CHAR etc.

DATATYPE	**Description**
CHAR	string(0-255), string with size = (0, 255], e.g., CHAR(251)
VARCHAR	string(0-255)
TINYTEXT	String(0-255)
TEXT	string(0-65535)
BLOB	string(0-65535)
MEDIUMTEXT	string(0-16777215)
MEDIUMBLOB	string(0-16777215)
LONGTEXT	string(0-4294967295)
LONGBLOB	string(0-4294967295)
TINYINT	integer(-128 to 127)
SMALLINT	integer(-32768 to 32767)
MEDIUMINT	integer(-8388608 to 8388607)
INT	integer(-2147483648 to 2147483647)
BIGINT	integer (-9223372036854775808 to 9223372036854775807)
FLOAT	Decimal with precision to 23 digits
DOUBLE	Decimal with 24 to 53 digits

DATATYPE	Description
DECIMAL	Double stored as string
DATE	YYYY-MM-DD
DATETIME	YYYY-MM-DD HH:MM:SS
TIMESTAMP	YYYYMMDDHHMMSS
TIME	HH:MM:SS
ENUM	One of the preset values
SET	One or many of the preset values
BOOLEAN	0/1
BIT	e.g., BIT(n), n upto 64, store values in bits.

3. Size: TINY < SMALL < MEDIUM < INT < BIGINT.
4. **Variable length Data types** e.g., VARCHAR, are better to use as they occupy space equal to the actual data size.
5. Values can also be unsigned e.g., INT UNSIGNED.
6. **Types of SQL commands:**
    1. **DDL** (data definition language): defining relation schema.
        1. **CREATE**: create table, DB, view.
        2. **ALTER TABLE**: modification in table structure. e.g, change column datatype or add/remove columns.
        3. **DROP**: delete table, DB, view.
        4. **TRUNCATE**: remove all the tuples from the table.
        5. **RENAME**: rename DB name, table name, column name etc.
    2. **DRL/DQL** (data retrieval language / data query language): retrieve data from the tables.
        1. **SELECT**
    3. **DML** (data modification language): use to perform modifications in the DB
        1. **INSERT**: insert data into a relation
        2. **UPDATE**: update relation data.
        3. **DELETE**: delete row(s) from the relation.
    4. **DCL** (Data Control language): grant or revoke authorities from user.
        1. **GRANT**: access privileges to the DB
        2. **REVOKE**: revoke user access privileges.
    5. **TCL** (Transaction control language): to manage transactions done in the DB
        1. **START TRANSACTION**: begin a transaction
        2. **COMMIT**: apply all the changes and end transaction
        3. **ROLLBACK**: discard changes and end transaction
        4. **SAVEPOINT**: checkout within the group of transactions in which to rollback.

MANAGING DB (DDL)

1. **Creation of DB**
    1. **CREATE DATABASE IF NOT EXISTS db-name;**
    2. **USE** db-name; //need to execute to choose on which DB CREATE TABLE etc. commands will be executed.
       //make switching between DBs possible.
    3. **DROP** DATABASE IF EXISTS db-name; //dropping database.
    4. **SHOW** DATABASES; //list all the DBs in the server.
    5. **SHOW** TABLES; //list tables in the selected DB.

DATA RETRIEVAL LANGUAGE (DRL)

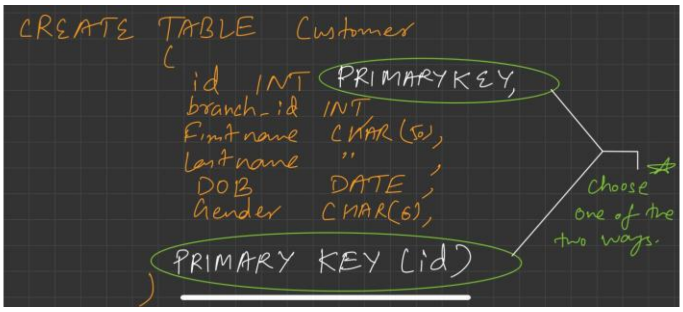
1. Syntax: SELECT <set of column names> FROM <table_name>;
2. Order of execution from RIGHT to LEFT.
3. Q. Can we use SELECT keyword without using FROM clause?
    1. Yes, using DUAL Tables.
    2. Dual tables are dummy tables created by MySQL, help users to do certain obvious actions without referring to user defined tables.
    3. e.g., SELECT
       55 + 11;
       SELECT now();
       SELECT ucase(); etc.
4. **WHERE**
    1. Reduce rows based on given conditions.
    2. E.g., SELECT * FROM customer WHERE age > 18;
5. **BETWEEN**
    1. SELECT * FROM customer WHERE age between 0 **AND** 100;
    2. In the above e.g., 0 and 100 are inclusive.
6. **IN**
    1. Reduces **OR** conditions;
    2. e.g., SELECT * FROM officers WHERE officer_name IN ('Lakshay', 'Maharana Pratap', 'Deepika');
7. **AND/OR/NOT**
    1. AND: WHERE cond1 AND cond2
    2. OR: WHERE cond1 OR cond2
    3. NOT: WHERE col_name NOT IN (1,2,3,4);
8. **IS NULL**
    1. e.g., SELECT * FROM customer WHERE prime_status is NULL;
9. **PaUern Searching / Wildcard ('%', '_')**
    1. '%', any number of character from 0 to n. Similar to '*' asterisk in regex.
    2. '_', only one character.
    3. SELECT * FROM customer WHERE name **LIKE** '%p_';
10. **ORDER BY**
    1. Sorting the data retrieved using **WHERE** clause.
    2. ORDER BY <column-name> DESC;
    3. DESC = Descending and ASC = Ascending
    4. e.g., SELECT * FROM customer ORDER BY name DESC;
11. **GROUP BY**
    1. GROUP BY Clause is used to collect data from multiple records and group the result by one or more column. It is generally used in a SELECT statement.
    2. Groups into category based on column given.
    3. SELECT c1, c2, c3 FROM sample_table WHERE cond GROUP BY c1, c2, c3.
    4. All the column names mentioned after SELECT statement shall be repeated in GROUP BY, in order to successfully execute the query.
    5. Used with aggregation functions to perform various actions.
        1. COUNT()
        2. SUM()
        3. AVG()
        4. MIN()
        5. MAX()
12. **DISTINCT**
    1. Find distinct values in the table.
    2. SELECT DISTINCT(col_name) FROM table_name;
    3. GROUP BY can also be used for the same
        1. "Select col_name from table GROUP BY col_name;" same output as above DISTINCT query.

    2. SQL is smart enough to realise that if you are using GROUP BY and not using any aggregation function, then you mean “DISTINCT”.

13. **GROUP BY HAVING**
    1. Out of the categories made by GROUP BY, we would like to know only particular thing (cond).
    2. Similar to WHERE.
    3. Select COUNT(cust_id), country from customer GROUP BY country HAVING COUNT(cust_id) > 50;
    4. WHERE vs HAVING
        1. Both have same function of filtering the row base on certain conditions.
        2. WHERE clause is used to filter the rows from the table based on specified condition
        3. HAVING clause is used to filter the rows from the groups based on the specified condition.
        4. HAVING is used after GROUP BY while WHERE is used before GROUP BY clause.
        5. If you are using HAVING, GROUP BY is necessary.
        6. WHERE can be used with SELECT, UPDATE G DELETE keywords while GROUP BY used with SELECT.

CONSTRAINTS (DDL)

1. **Primary Key**



```
CREATE TABLE Customer
  (
    id        INT   PRIMARYKEY,
    branch_id INT
    Firstname  CHAR(50),
    Lastname   ''      ,
    DOB        DATE    ,
    Gender    CHAR(6),
   PRIMARY KEY (id)
  )
```

    1. PK is not null, unique and only one per table.

2. **Foreign Key**
    1. FK refers to PK of other table.
    2. Each relation can having any number of FK.
    3. CREATE TABLE
       ORDER (id INT
       PRIMARY KEY,
       delivery_date DATE,
       order_placed_date
       DATE, cust_id INT,
       FOREIGN KEY (cust_id) REFERENCES customer(id)
       );

3. **UNIQUE**
    1. Unique, can be null, table can have multiple unique attributes.
    2. CREATE TABLE customer (
       …
       email VARCHAR(1024) UNIQUE,
       …
       );

4. **CHECK**
    1. CREATE TABLE customer (
       …
       CONSTRAINT age_check CHECK (age > 12),
       …
       );
    2. “age_check”, can also avoid this, MySQL generates name of constraint automatically.

5. **DEFAULT**
    1. Set default value of the column.
    2. CREATE TABLE account (
       ...
       saving-rate DOUBLE NOT NULL DEFAULT 4.25,
       ...
       );
6. An attribute can be **PK and FK both** in a table.
7. **ALTER OPERATIONS**
    1. Changes schema
    2. **ADD**
        1. **Add new column.**
        2. ALTER TABLE table_name ADD new_col_name datatype ADD new_col_name_2 datatype;
        3. e.g., ALTER TABLE customer ADD age INT NOT NULL;
    3. **MODIFY**
        1. **Change datatype of an attribute.**
        2. ALTER TABLE table-name MODIFY col-name col-datatype;
        3. E.g., VARCHAR TO CHAR
           ALTER TABLE customer MODIFY name CHAR(1024);
    4. **CHANGE COLUMN**
        1. **Rename column name.**
        2. ALTER TABLE table-name CHANGE COLUMN old-col-name new-col-name new-col-datatype;
        3. e.g., ALTER TABLE customer CHANGE COLUMN name customer-name VARCHAR(1024);
    5. **DROP COLUMN**
        1. **Drop a column completely.**
        2. ALTER TABLE table-name DROP COLUMN col-name;
        3. e.g., ALTER TABLE customer DROP COLUMN middle-name;
    6. **RENAME**
        1. **Rename table name itself.**
        2. ALTER TABLE table-name RENAME TO new-table-name;
        3. e.g., ALTER TABLE customer RENAME TO customer-details;

DATA MANIPULATION LANGUAGE (DML)

1. **INSERT**
    1. INSERT INTO table-name(col1, col2, col3) VALUES (v1, v2, v3), (val1, val2, val3);
2. **UPDATE**
    1. UPDATE table-name SET col1 = 1, col2 = 'abc' WHERE id = 1;
    2. Update multiple rows e.g.,
        1. UPDATE student SET standard = standard + 1;
    3. **ON UPDATE CASCADE**
        1. Can be added to the table while creating constraints. Suppose there is a situation where we have two tables such that primary key of one table is the foreign key for another table. if we update the primary key of the first table then using the ON UPDATE CASCADE foreign key of the second table automatically get updated.
3. **DELETE**
    1. DELETE FROM table-name WHERE id = 1;
    2. DELETE FROM table-name; //all rows will be deleted.
    3. **DELETE CASCADE - (to overcome DELETE constraint of Referential constraints)**
        1. What would happen to child entry if parent table's entry is deleted?
        2. CREATE TABLE ORDER (
           order_id int
           PRIMARY KEY,
           delivery_date DATE,
           cust_id INT,

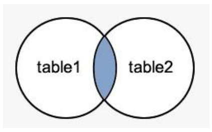
FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE CASCADE
);

3. **ON DELETE NULL - (can FK have null values?)**
   1. CREATE TABLE ORDER (
      order_id int
      PRIMARY KEY,
      delivery_date DATE,
      cust_id INT,
      FOREIGN KEY(cust_id) REFERENCES customer(id) ON DELETE SET NULL
      );

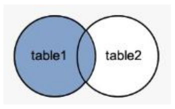
4. **REPLACE**
   1. Primarily used for already present tuple in a table.
   2. As UPDATE, using REPLACE with the help of WHERE clause in PK, then that row will be replaced.
   3. As INSERT, if there is no duplicate data new tuple will be inserted.
   4. REPLACE INTO student (id, class) VALUES(4, 3);
   5. REPLACE INTO table SET col1 = val1, col2 = val2;

JOINING TABLES

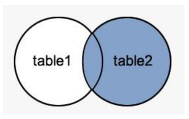
1. All **RDBMS** are relational in nature, we refer to other tables to get meaningful outcomes.
2. FK are used to do reference to other table.
3. **INNER JOIN**
   1. Returns a resultant table that has matching values from both the tables or all the tables.
   2. SELECT column-list FROM table1 INNER JOIN table2 ON condition1 INNER JOIN table3 ON condition2
      ...;



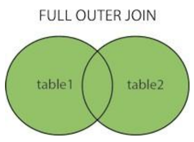
   3. **Alias in MySQL (AS)**
      1. Aliases in MySQL is used to give a temporary name to a table or a column in a table for the purpose of a particular query. It works as a nickname for expressing the tables or column names. It makes the query short and neat.
      2. SELECT col_name AS alias_name FROM table_name;
      3. SELECT col_name1, col_name2,... FROM table_name AS alias_name;
4. **OUTER JOIN**
   1. **LEFT JOIN**
      1. This returns a resulting table that all the data from lei table and the matched data from the right table.
      2. SELECT columns FROM table LEFT JOIN table2 ON Join_Condition;



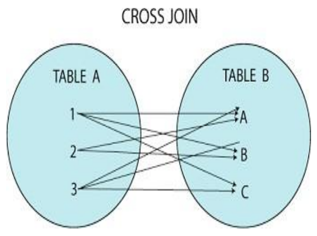
   2. **RIGHT JOIN**
      1. This returns a resulting table that all the data from right table and the matched data from the lei table.
      2. SELECT columns FROM table RIGHT JOIN table2 ON join_cond;



   3. **FULL JOIN**
      1. This returns a resulting table that contains all data when there is a match on lei or right table data.
      2. **Emulated** in MySQL using LEFT and RIGHT JOIN.
      3. LEFT JOIN UNION RIGHT JOIN.
      4. SELECT columns FROM table1 as t1 LEFT JOIN table2 as t2 ON t1.id = t2.id
         UNION
      SELECT columns FROM table1 as t1 RIGHT JOIN table2 as t2 ON t1.id = t2.id;
      5. UNION ALL, can also be used this will duplicate values as well while UNION gives unique values.



5. **CROSS JOIN**
   1. This returns all the cartesian products of the data present in both tables. Hence, all possible variations are reflected in the output.
   2. Used rarely in practical purpose.
   3. Table-1 has 10 rows and table-2 has 5, then resultant would have 50 rows.
   4. SELECT column-lists FROM table1 CROSS JOIN table2;



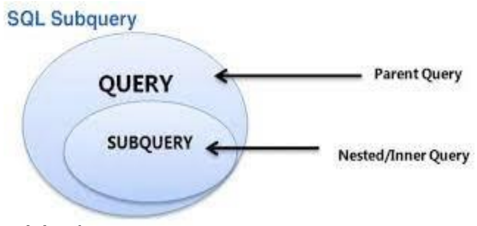
6. **SELF JOIN**
    1. It is used to get the output from a particular table when the same table is joined to itself.
    2. Used very less.
    3. Emulated using INNER JOIN.
    4. SELECT columns FROM table as t1 INNER JOIN table as t2 ON t1.id = t2.id;
7. **Join without using join keywords.**
    1. SELECT * FROM table1, table2 WHERE condition;
    2. e.g., SELECT artist_name, album_name, year_recordedFROM artist, albumWHERE artist.id = album.artist_id;

SET OPERATIONS
1. Used to combine multiple select statements.
2. Always gives distinct rows.

JOIN	**SET Operations**
Combines multiple tables based on matching condition.	Combination is resulting set from two or more SELECT statements.
Column wise combination.	Row wise combination.
Data types of two tables can be different.	Datatypes of corresponding columns from each table should be the same.
Can generate both distinct or duplicate rows.	Generate distinct rows.
The number of column(s) selected may or may not be the same from each table.	The number of column(s) selected must be the same from each table.
Combines results horizontally.	Combines results vertically.

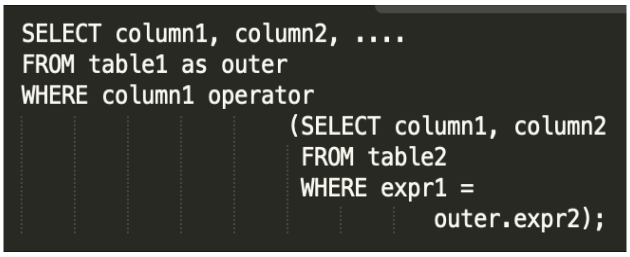
3. **UNION**
    1. Combines two or more SELECT statements.
    2. SELECT * FROM
       table1 UNION
       SELECT * FROM table2;
    3. Number of column, order of column must be same for table1 and table2.
4. **INTERSECT**
    1. Returns common values of the tables.
    2. Emulated.
    3. SELECT DISTINCT column-list FROM table-1 INNER JOIN table-2 USING(join_cond);
    4. SELECT DISTINCT * FROM table1 INNER JOIN table2 ON USING(id);
5. **MINUS**
    1. This operator returns the distinct row from the first table that does not occur in the second table.
    2. Emulated.
    3. SELECT column_list FROM table1 LEFT JOIN table2 ON condition WHERE table2.column_name IS NULL;
    4. e.g., SELECT id FROM table-1 LEFT JOIN table-2 USING(id) WHERE table-2.id IS NULL;

SUB QUERIES



1. Outer query depends on inner query.
2. Alternative to joins.
3. Nested queries.
4. SELECT column_list (s) FROM table_name WHERE column_name OPERATOR (SELECT column_list (s) FROM table_name [WHERE]);
5. e.g., SELECT * FROM table1 WHERE col1 IN (SELECT col1 FROM table1);
6. Sub queries exist mainly in 3 clauses
    1. Inside a WHERE clause.

    2. Inside a FROM clause.
    3. Inside a SELECT clause.
7. **Subquery using FROM clause**
    1. SELECT MAX(rating) FROM (SELECT * FROM movie WHERE country = ‘India’) as temp;
8. **Subquery using SELECT**
    1. SELECT (SELECT column_list(s) FROM T_name WHERE condition), columnList(s) FROM T2_name WHERE condition;
9. **Derived Subquery**
    1. SELECT columnLists(s) FROM (SELECT columnLists(s) FROM table_name WHERE [condition]) as new_table_name;
10. **Co-related sub-queries**
    1. With a normal nested subquery, the inner SELECT query runs first and executes once, returning values to be used by the main query. A correlated subquery, however, executes once for each candidate row considered by the outer query. In other words, the inner query is driven by the outer query.



```
SELECT column1, column2, ....
FROM table1 as outer
WHERE column1 operator
                    (SELECT column1, column2
                     FROM table2
                     WHERE expr1 =
                               outer.expr2);
```

JOIN VS SUB-QUERIES

JOINS	**SUBQUERIES**
Faster	Slower
Joins maximise calculation burden on DBMS	Keeps responsibility of calculation on user.
Complex, difficult to understand and implement	Comparatively easy to understand and implement.
Choosing optimal join for optimal use case is difficult	Easy.

MySQL VIEWS

1. A view is a database object that has no values. Its contents are based on the base table. It contains rows and columns similar to the real table.
2. In MySQL, the View is a **virtual table** created by a query by joining one or more tables. It is operated similarly to the base table but does not contain any data of its own.
3. The View and table have one main difference that the views are definitions built on top of other tables (or views). If any changes occur in the underlying table, the same changes reflected in the View also.
4. CREATE VIEW view_name AS SELECT columns FROM tables [WHERE conditions];
5. ALTER VIEW view_name AS SELECT columns FROM table WHERE conditions;
6. DROP VIEW IF EXISTS view_name;
7. CREATE VIEW Trainer AS SELECT c.course_name, c.trainer, t.email FROM courses c, contact t WHERE c.id = t.id; (View using Join clause).

NOTE: We can also import/export table schema from files (.csv or json).

Normalisation

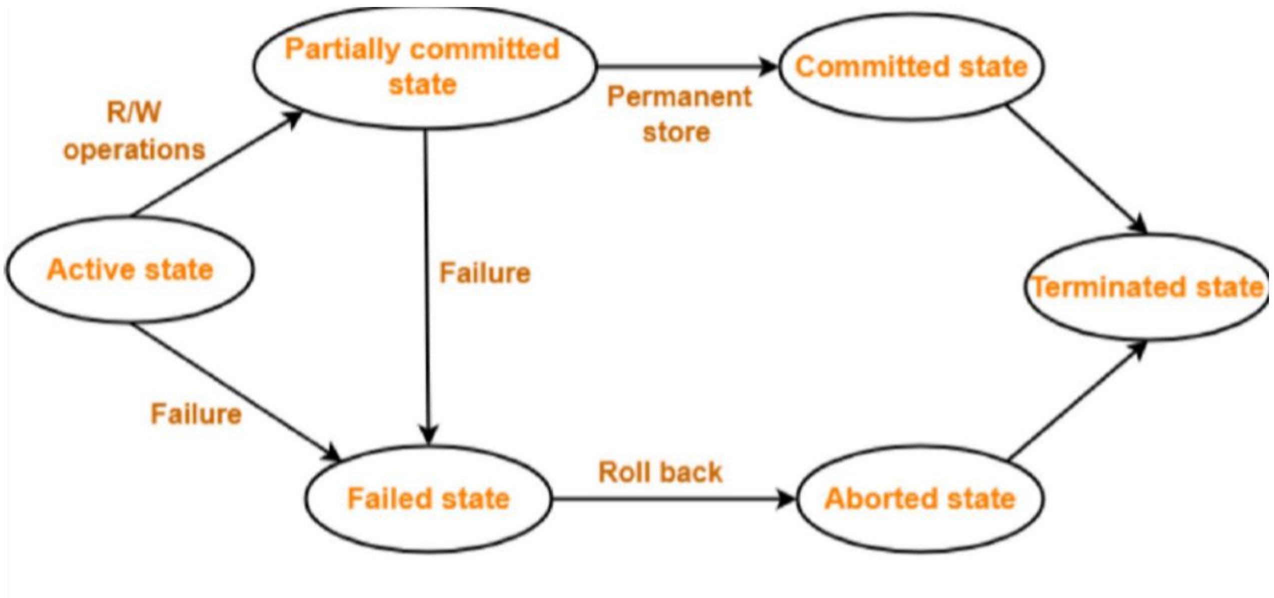
1. **Normalisation** is a step towards DB optimisation.
2. **Functional Dependency (FD)**
    1. It's a relationship between the primary key attribute (usually) of the relation to that of the other attribute of the relation.
    2. X -> Y, the lei side of FD is known as a **Determinant**, the right side of the production is known as a **Dependent**.
    3. **Types of FD**
        1. **Trivial FD**
            1. A -> B has trivial functional dependency if B is a subset of A. A->A, B->B are also Trivial FD.
        2. **Non-trivial FD**
            1. A -> B has a non-trivial functional dependency if B is not a subset of A. [A intersection B is NULL].
    4. **Rules of FD (Armstrong's axioms)**
        1. **Reflexive**
            1. If 'A' is a set of attributes and 'B' is a subset of 'A'. Then, A-> B holds.
            2. If A ⊇ B then A -> B.
        2. **Augmentation**
            1. If B can be determined from A, then adding an attribute to this functional dependency won't change anything.
            2. If A-> B holds, then AX-> BX holds too. 'X' being a set of attributes.
        3. **Transitivity**
            1. If A determines B and B determines C, we can say that A determines C.
            2. if A-> B and B-> C then A-> C.
3. **Why Normalisation?**
    1. To avoid redundancy in the DB, not to store redundant data.
4. **What happen if we have redundant data?**
    1. Insertion, deletion and updation anomalies arises.
5. **Anomalies**
    1. Anomalies means abnormalities, there are three types of anomalies introduced by data redundancy.
    2. **Insertion anomaly**
        1. When certain data (attribute) can not be inserted into the DB without the presence of other data.
    3. **Deletion anomaly**
        1. The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
    4. **Updation anomaly** (or modification anomaly)
        1. The update anomaly is when an update of a single data value requires multiple rows of data to be updated.
        2. Due to updation to many places, may be **Data inconsistency** arises, if one forgets to update the data at all the intended places.
    5. Due to these anomalies, **DB size increases** and **DB performance become very slow.**
    6. To rectify these anomalies and the effect of these of DB, we use **<u>Database optimisation technique</u>** called
       NORMALISATION.
6. **What is Normalisation?**
    1. Normalisation is used to minimise the redundancy from a relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
    2. Normalisation divides the composite attributes into individual attributes OR larger table into smaller and links them using relationships.
    3. The normal form is used to reduce redundancy from the database table.
7. **Types of Normal forms**
    1. **1NF**
        1. Every relation cell must have atomic value.
        2. Relation must not have multi-valued attributes.

2. **2NF**
    1. Relation must be in 1NF.
    2. There should not be any partial dependency.
        1. All non-prime attributes must be fully dependent on PK.
        2. Non prime attribute can not depend on the part of the PK.
3. **3NF**
    1. Relation must be in 2NF.
    2. No transitivity dependency exists.
        1. Non-prime attribute should not find a non-prime attribute.
4. **BCNF (Boyce-Codd normal form)**
    1. Relation must be in 3NF.
    2. FD: A -> B, A must be a super key.
        1. We must not derive prime attribute from any prime or non-prime attribute.

8. **Advantages of Normalisation**
    1. Normalisation helps to minimise data redundancy.
    2. Greater overall database organisation.
    3. Data consistency is maintained in DB.

Transaction

1. **Transaction**
    1. A unit of work done against the DB in a logical sequence.
    2. Sequence is very important in transaction.
    3. It is a logical unit of work that contains one or more SQL statements. The result of all these statements in a transaction either gets completed successfully (all the changes made to the database are permanent) or if at any point any failure happens it gets rollbacked (all the changes being done are undone.)
2. **ACID Properties**
    1. To ensure integrity of the data, we require that the DB system maintain the following properties of the transaction.
    2. **Atomicity**
        1. Either all operations of transaction are reflected properly in the DB, or none are.
    3. **Consistency**
        1. Integrity constraints must be maintained before and after transaction.
        2. DB must be consistent after transaction happens.
    4. **Isolation**
        1. Even though multiple transactions may execute concurrently, the system guarantees that, for every pair of transactions Ti and Tj, it appears to Ti that either Tj finished execution before Ti started, or Tj started execution after Ti finished. Thus, each transaction is unaware of other transactions executing concurrently in the system.
        2. Multiple transactions can happen in the system in isolation, without interfering each other.
    5. **Durability**
        1. After transaction completes successfully, the changes it has made to the database persist, even if there are system failures.
3. **Transaction states**



Transaction States in DBMS

    1. **Active state**
        1. The very first state of the life cycle of the transaction, all the read and write operations are being performed. If they execute without any error the T comes to Partially committed state. Although if any error occurs then it leads to a Failed state.
    2. **Partially committed state**
        1. After transaction is executed the changes are saved in the buffer in the main memory. If the changes made are permanent on the DB then the state will transfer to the committed state and if there is any failure, the T will go to Failed state.

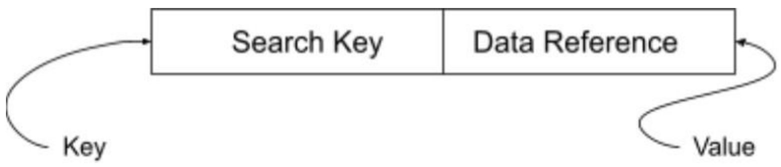
3. **Committed state**
    1. When updates are made permanent on the DB. Then the T is said to be in the committed state. Rollback can't be done from the committed states. New consistent state is achieved at this stage.
4. **Failed state**
    1. When T is being executed and some failure occurs. Due to this it is impossible to continue the execution of the T.
5. **Aborted state**
    1. When T reaches the failed state, all the changes made in the buffer are reversed. After that the T rollback completely. T reaches abort state after rollback. DB's state prior to the T is achieved.
6. **Terminated state**
    1. A transaction is said to have terminated if has either committed or aborted.

How to implement Atomicity and Durability in Transactions

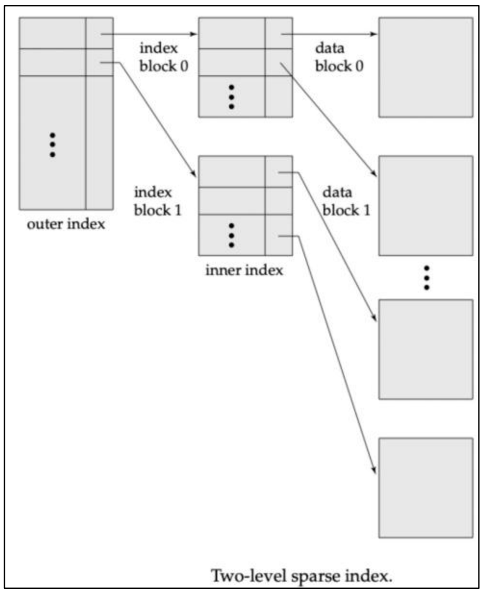
1. **Recovery Mechanism Component** of DBMS supports **atomicity and durability.**
2. **Shadow-copy scheme**
    1. Based on making copies of DB (aka, **shadow copies)**.
    2. Assumption only one Transaction (T) is active at a time.
    3. A pointer called **db-pointer** is maintained on the **disk**; which at any instant points to current copy of DB.
    4. T, that wants to update DB first creates a complete copy of DB.
    5. All further updates are done on new DB copy leaving the original copy (shadow copy) untouched.
    6. If at any point the **T has to be aborted** the system deletes the new copy. And the old copy is not affected.
    7. If T success, it is committed as,
        1. OS makes sure all the pages of the new copy of DB written on the disk.
        2. DB system updates the db-pointer to point to the new copy of DB.
        3. New copy is now the current copy of DB.
        4. The old copy is deleted.
        5. The T is said to have been **COMMITTED** at the point where the updated db-pointer is written to disk.
    8. **Atomicity**
        1. If T fails at any time before db-pointer is updated, the old content of DB are not affected.
        2. T abort can be done by just deleting the new copy of DB.
        3. Hence, either all updates are reflected or none.
    9. **Durability**
        1. Suppose, system fails are any time before the updated db-pointer is written to disk.
        2. When the system restarts, it will read db-pointer G will thus, see the original content of DB and none of the effects of T will be visible.
        3. T is assumed to be successful only when db-pointer is updated.
        4. If **system fails after** db-pointer has been updated. Before that all the pages of the new copy were written to disk. Hence, when system restarts, it will read new DB copy.
    10. The implementation is dependent on write to the db-pointer being atomic. Luckily, disk system provide atomic updates to entire block or at least a disk sector. So, we make sure db-pointer lies entirely in a single sector. By storing db-pointer at the beginning of a block.
    11. **Inefficient**, as entire DB is copied for every Transaction.
3. **Log-based recovery methods**
    1. The log is a sequence of records. Log of each transaction is maintained in some **stable storage** so that if any failure occurs, then it can be recovered from there.
    2. If any operation is performed on the database, then it will be recorded in the log.
    3. But the process of storing the logs should be done **before** the actual transaction is applied in the database.
    4. **Stable storage** is a classification of computer data storage technology that guarantees atomicity for any given write operation and allows software to be written that is robust against some hardware and power failures.
    5. **Deferred DB Modifications**
        1. Ensuring **atomicity** by recording all the DB modifications in the log but deferring the execution of all the write operations until the final action of the T has been executed.
        2. Log information is used to execute deferred writes when T is completed.
        3. If system crashed before the T completes, or if T is aborted, the information in the logs are ignored.
        4. If T completes, the records associated to it in the log file are used in executing the deferred writes.
        5. If failure occur while this updating is taking place, we preform redo.
    6. **Immediate DB Modifications**
        1. DB modifications to be output to the DB while the T is still in active state.
        2. DB modifications written by active T are called uncommitted modifications.
        3. In the event of crash or T failure, system uses old value field of the log records to restore modified values.
        4. Update takes place only after log records in a stable storage.
        5. Failure handling
            1. System failure before T completes, or if T aborted, then old value field is used to undo the T.
            2. If T completes and system crashes, then new value field is used to redo T having commit logs in the logs.

Indexing in DBMS

1. **Indexing** is used to **optimise the performance** of a database by minimising the number of disk accesses required when a query is processed.
2. The index is a type of **data structure.** It is used to locate and access the data in a database table quickly.
3. **Speeds up operation** with read operations like **SELECT** queries, **WHERE** clause etc.
4. **Search Key**: Contains copy of primary key or candidate key of the table or something else.
5. **Data Reference**: Pointer holding the address of disk block where the value of the corresponding key is stored.
6. Indexing is **optional**, but increases access speed. It is not the primary mean to access the tuple, it is the secondary mean.
7. **Index file is always sorted.**
8. **Indexing Methods**
    1. **Primary Index (Clustering Index)**
        1. A file may have several indices, on different search keys. If the data file containing the records is sequentially ordered, a Primary index is an index whose search key also defines the sequential order of the file.
        2. **NOTE**: The term primary index is sometimes used to mean an index on a primary key. However, such usage is **nonstandard** and **should be avoided.**
        3. All files are ordered sequentially on some search key. It could be Primary Key or non-primary key.
        4. **Dense And Sparse Indices**
            1. **Dense Index**
                1. The dense index contains an index record for every search key value in the data file.
                2. 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.
                3. It needs more space to store index record itself. The index records have the search key and a pointer to the actual record on the disk.
            2. **Sparse Index**
                1. An index record appears for only some of the search-key values.
                2. Sparse Index helps you to resolve the issues of dense Indexing in DBMS. In this method of indexing technique, a range of index columns stores the same data block address, and when data needs to be retrieved, the block address will be fetched.
        5. Primary Indexing can be based on Data file is sorted w.r.t Primary Key attribute or non-key attributes.
        6. **Based on Key attribute**
            1. Data file is sorted w.r.t primary key attribute.
            2. PK will be used as search-key in Index.
            3. Sparse Index will be formed i.e., no. of entries in the index file = no. of blocks in datafile.
        7. **Based on Non-Key attribute**
            1. Data file is sorted w.r.t non-key attribute.
            2. No. Of entries in the index = unique non-key attribute value in the data file.
            3. This is dense index as, all the unique values have an entry in the index file.
            4. E.g., Let's assume that a company recruited many employees in various departments. In this case, clustering indexing in DBMS should be created for all employees who belong to the same dept.
        8. **Multi-level Index**
            1. Index with two or more levels.
            2. If the single level index become enough



large that the binary search it self would take much time, we can break down indexing into multiple levels.



Two-level sparse index.

2. **Secondary Index (Non-Clustering Index)**
   1. Datafile is unsorted. Hence, Primary Indexing is not possible.
   2. Can be done on key or non-key attribute.
   3. Called secondary indexing because normally one indexing is already applied.
   4. No. Of entries in the index file = no. of records in the data file.
   5. It's an example of Dense index.

9. **Advantages of Indexing**
   1. Faster access and retrieval of data.
   2. IO is less.
10. **Limitations of Indexing**
    1. Additional space to store index table
    2. Indexing Decrease performance in INSERT, DELETE, and UPDATE query.