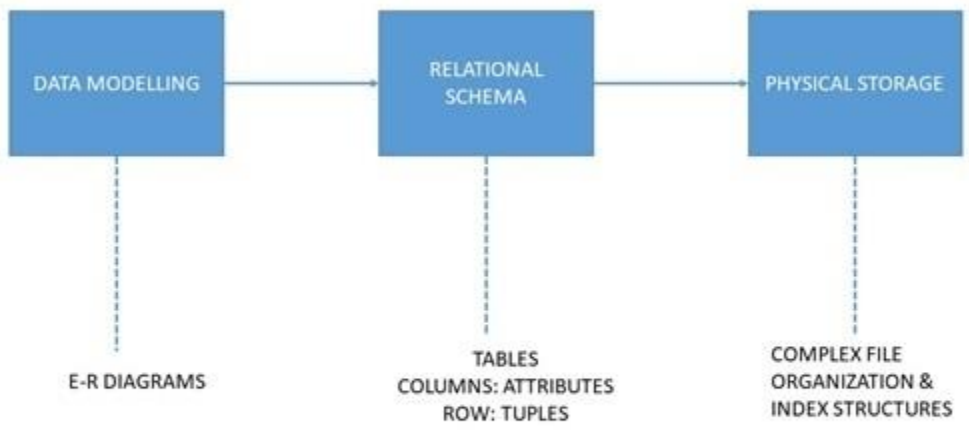
UNIT-3 Relational Database Design

The relational data model was introduced by C. F. Codd in 1970. Currently, it is the most widely used data model. The relational data model describes the world as "a collection of inter-related relations (or tables)." A relational data model involves the use of data tables that collect groups of elements into relations. These models work based on the idea that each table setup will include a primary key or identifier. Other tables use that identifier to provide "relational" data links and results.



As mentioned, the primary key is a fundamental tool in creating and using relational data models. It must be unique for each member of a data set. It must be populated for all members. Inconsistencies can cause problems in how developers retrieve data. Other issues with relational database designs include excessive duplication of data, faulty or partial data, or improper links or associations between tables. A large part of routine database administration involves evaluating all the data sets in a database to make sure that they are consistently populated and will respond well to SQL or any other data retrieval method.

For example, a conventional database row would represent a tuple, which is a set of data that revolves around an instance or virtual object so that the primary key is its unique identifier. A column name in a data table is associated with an attribute, an identifier or feature that all parts of a data set have. These and other strict conventions help to provide database administrators and designers with standards for crafting relational database setups.

Database Design Objective

- **Eliminate Data Redundancy:** the same piece of data shall not be stored in more than one place. This is because duplicate data not only waste storage spaces but also easily lead to inconsistencies.
- **Ensure Data Integrity and Accuracy:** is the maintenance of, and the assurance of the accuracy and consistency of, data over its entire life-cycle, and is a critical aspect to the design, implementation, and usage of any system which stores, processes, or retrieves data.

The relational model has provided the basis for:

- Research on the theory of data/relationship/constraint
- Numerous database design methodologies
- The standard database access language called structured query language (SQL)
- Almost all modern commercial database management systems

Relational databases go together with the development of SQL. The simplicity of SQL - where even a novice can learn to perform basic queries in a short period of time - is a large part of the reason for the popularity of the relational model.

The two tables below relate to each other through the product code field. Any two tables can relate to each other simply by creating a field they have in common.

Table 1

Product_code	Description	Price
A416	Colour Pen	₹ 25.00
C923	Pencil box	₹ 45.00

Table 2

Invoice_code	Invoice_line	Product_code	Quantity
3804	1	A416	15
3804	2	C923	24

There are four stages of an RDM which are as follows −

- **Relations and attributes** − The various tables and attributes related to each table are identified. The tables represent entities, and the attributes represent the properties of the respective entities.
- **Primary keys** − The attribute or set of attributes that help in uniquely identifying a record is identified and assigned as the primary key.
- **Relationships** −The relationships between the various tables are established with the help of foreign keys. Foreign keys are attributes occurring in a table that are primary keys of another table. The types of relationships that can exist between the relations (tables) are One to one, One to many, and Many to many
- **Normalization** − This is the process of optimizing the database structure. Normalization simplifies the database design to avoid redundancy and confusion. The different normal forms are as follows:

1. First normal form 2. Second normal form 3. Third normal form 4. Boyce-Codd normal form 5. Fifth normal form

By applying a set of rules, a table is normalized into the above normal forms in a linearly progressive fashion. The efficiency of the design gets better with each higher degree of normalization.

Advantages of Relational Databases

The main advantages of relational databases are that they enable users to easily categorize and store data that can later be queried and filtered to extract specific information for reports. Relational databases are also easy to extend and aren't reliant on the physical organization. After the original database creation, a new data category can be added without all existing applications being modified.

Other Advantages

- **Accurate** − Data is stored just once, which eliminates data deduplication.
- **Flexible** − Complex queries are easy for users to carry out.
- **Collaborative** −Multiple users can access the same database.
- **Trusted** −Relational database models are mature and well-understood.
- **Secure** − Data in tables within relational database management systems (RDBMS) can be limited to allow access by only particular users.

Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

1. X → Y

The left side of FD is known as a determinant, the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

1. Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

Types of Functional dependencies in DBMS:

1. Trivial functional dependency
2. Non-Trivial functional dependency

3. Multivalued functional dependency
4. Transitive functional dependency

1. Trivial Functional Dependency

In **Trivial Functional Dependency**, a dependent is always a subset of the determinant.
i.e. If **X → Y** and **Y is the subset of X**, then it is called trivial functional dependency

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **{roll_no, name} → name** is a trivial functional dependency, since the dependent **name** is a subset of determinant set **{roll_no, name}**
Similarly, **roll_no → roll_no** is also an example of trivial functional dependency.

2. Non-trivial Functional Dependency

In **Non-trivial functional dependency**, the dependent is strictly not a subset of the determinant.
i.e. If **X → Y** and **Y is not a subset of X**, then it is called Non-trivial functional dependency.

For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18

Here, **roll_no → name** is a non-trivial functional dependency, since the dependent **name** is **not a subset of** determinant **roll_no**
Similarly, **{roll_no, name} → age** is also a non-trivial functional dependency, since **age** is **not a subset of {roll_no, name}**

3. Multivalued Functional Dependency

In **Multivalued functional dependency**, entities of the dependent set are **not dependent on each other.**

i.e. If **a → {b, c}** and there exists **no functional dependency** between **b and c**, then it is called a **multivalued functional dependency.**
For example,

roll_no	name	age
42	abc	17
43	pqr	18
44	xyz	18
45	abc	19

Here, **roll_no → {name, age}** is a multivalued functional dependency, since the dependents **name** & **age** are **not dependent** on each other(i.e. **name → age** or **age → name doesn't exist !**)

5. ***Transitive Functional Dependency***

In transitive functional dependency, dependent is indirectly dependent on determinant.
i.e. If **a → b** & **b → c**, then according to axiom of transitivity, **a → c**. This is a **transitive functional dependency**
For example,

enrol_no	name	dept	building_no
42	abc	CO	4
43	pqr	EC	2
44	xyz	IT	1
45	abc	EC	2

Here, **enrol_no → dept** and **dept → building_no**,
Hence, according to the axiom of transitivity, **enrol_no → building_no** is a valid

functional dependency. This is an indirect functional dependency, hence called Transitive functional dependency.

Normalization

A large database defined as a single relation may result in data duplication. This repetition of data may result in:

- Making relations very large.
- It isn't easy to maintain and update data as it would involve searching many records in relation.
- Wastage and poor utilization of disk space and resources.
- The likelihood of errors and inconsistencies increases.

So to handle these problems, we should analyze and decompose the relations with redundant data into smaller, simpler, and well-structured relations that are satisfy desirable properties. Normalization is a process of decomposing the relations into relations with fewer attributes.

What is Normalization?

- Normalization is the process of organizing the data in the database.
- Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- Normalization divides the larger table into smaller and links them using relationships.
- The normal form is used to reduce redundancy from the database table.

Why do we need Normalization?

The main reason for normalizing the relations is removing these anomalies. Failure to eliminate anomalies leads to data redundancy and can cause data integrity and other problems as the database grows. Normalization consists of a series of guidelines that helps to guide you in creating a good database structure.

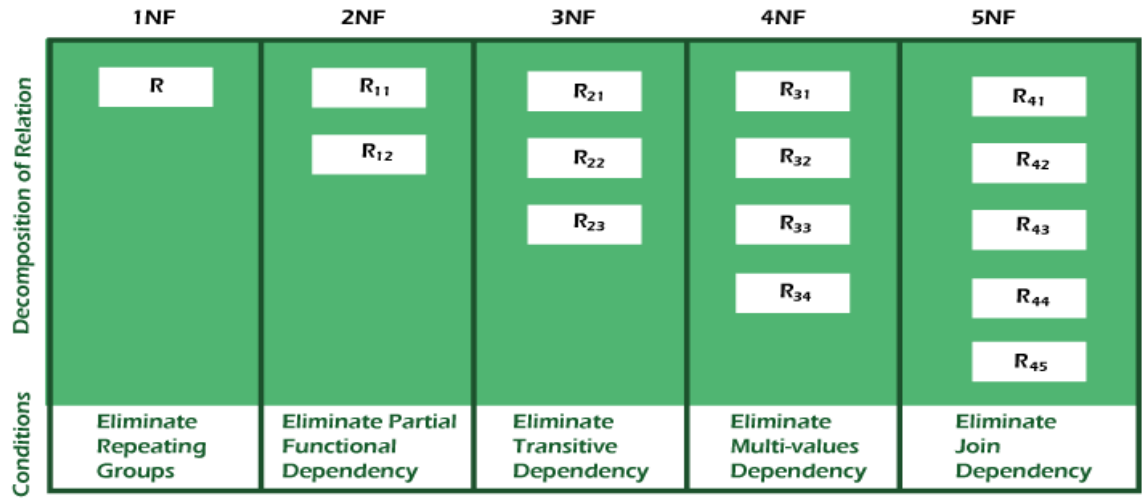
Data modification anomalies can be categorized into three types:

- **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
- **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
- **Updatation Anomaly:** The update anomaly is when an update of a single data value requires multiple rows of data to be updated.

Types of Normal Forms:

Normalization works through a series of stages called Normal forms. The normal forms apply to individual relations. The relation is said to be in particular normal form if it satisfies constraints.

Following are the various types of Normal forms:



Normal Form	Description
1NF	A relation is in 1NF if it contains an atomic value.
2NF	A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.
3NF	A relation will be in 3NF if it is in 2NF and no transition dependency exists.
BCNF	A stronger definition of 3NF is known as Boyce Codd's normal form.

4NF	A relation will be in 4NF if it is in Boyce Codd's normal form and has no multi-valued dependency.
5NF	A relation is in 5NF. If it is in 4NF and does not contain any join dependency, joining should be lossless.

Advantages of Normalization

- Normalization helps to minimize data redundancy.
- Greater overall database organization.
- Data consistency within the database.
- Much more flexible database design.
- Enforces the concept of relational integrity.

Disadvantages of Normalization

- You cannot start building the database before knowing what the user needs.
- The performance degrades when normalizing the relations to higher normal forms, i.e., 4NF, 5NF.
- It is very time-consuming and difficult to normalize relations of a higher degree.
- Careless decomposition may lead to a bad database design, leading to serious problems.

Keys

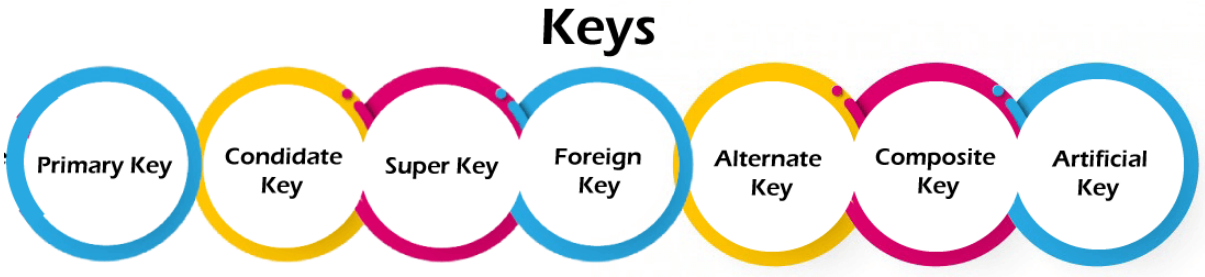
- Keys play an important role in the relational database.
- It is used to uniquely identify any record or row of data from the table. It is also used to establish and identify relationships between tables.

For example, ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.

STUDENT
ID
Name
Address
Course

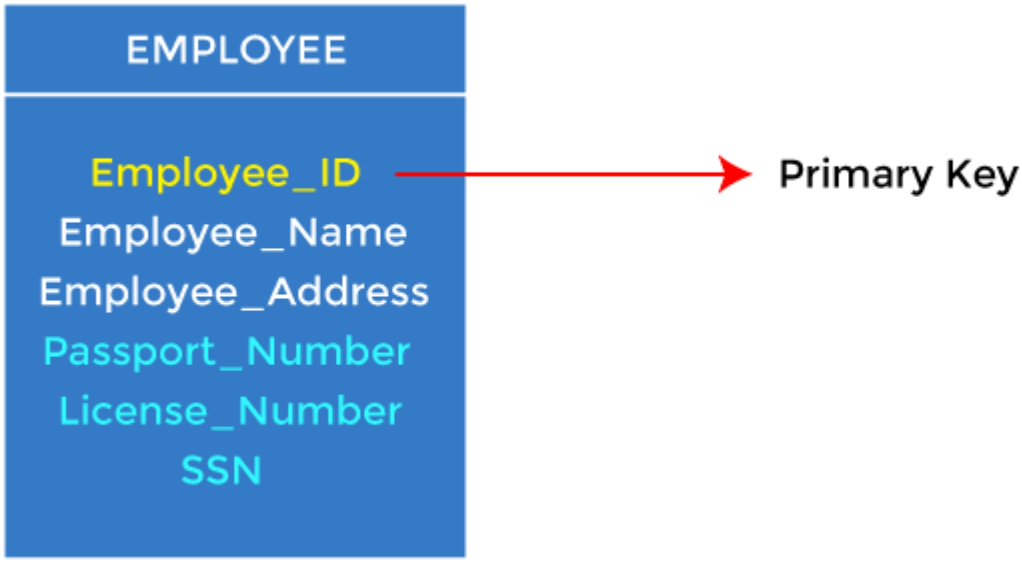
PERSON
Name
DOB
Passport, Number
License_Number
SSN

Types of keys:



1. Primary key

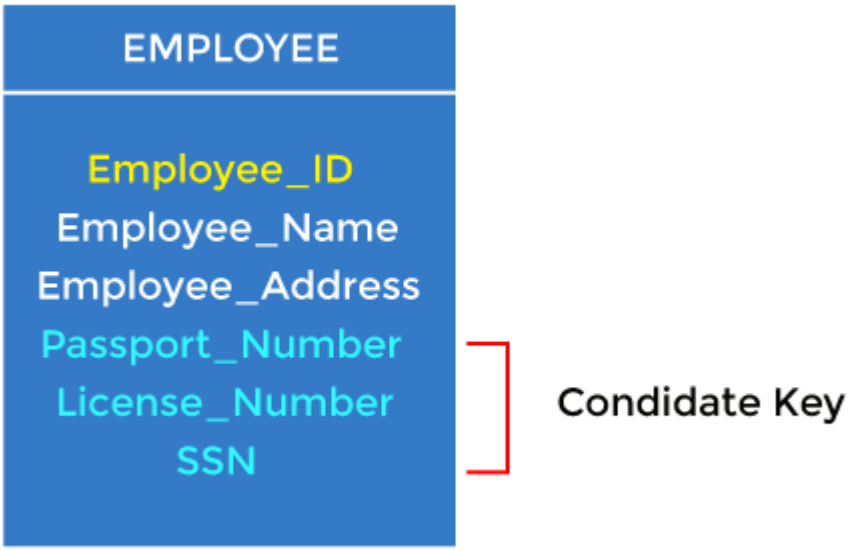
- It is the first key used to identify one and only one instance of an entity uniquely. An entity can contain multiple keys, as we saw in the PERSON table. The key which is most suitable from those lists becomes a primary key.
- In the EMPLOYEE table, ID can be the primary key since it is unique for each employee. In the EMPLOYEE table, we can even select License_Number and Passport_Number as primary keys since they are also unique.
- For each entity, the primary key selection is based on requirements and developers.



2. Candidate key

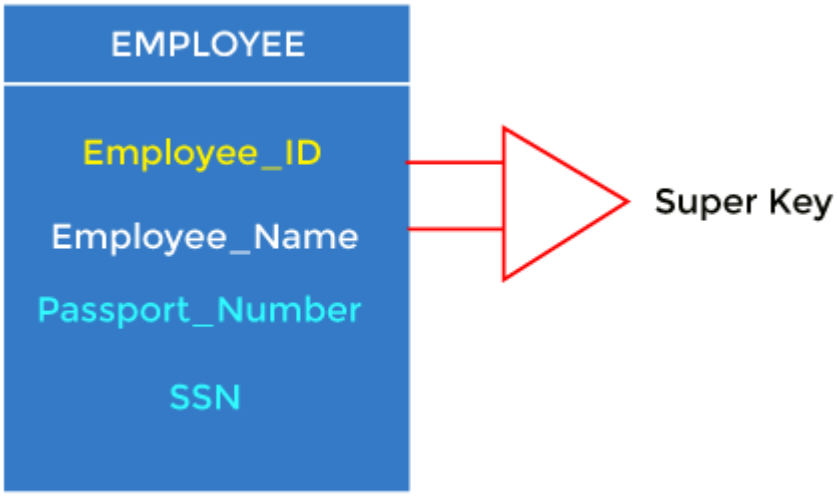
- A candidate key is an attribute or set of attributes that can uniquely identify a tuple.
- Except for the primary key, the remaining attributes are considered a candidate key. The candidate keys are as strong as the primary key.

For example: In the EMPLOYEE table, id is best suited for the primary key. The rest of the attributes, like SSN, Passport_Number, License_Number, etc., are considered a candidate key.



3. Super Key

Super key is an attribute set that can uniquely identify a tuple. A super key is a superset of a candidate key.

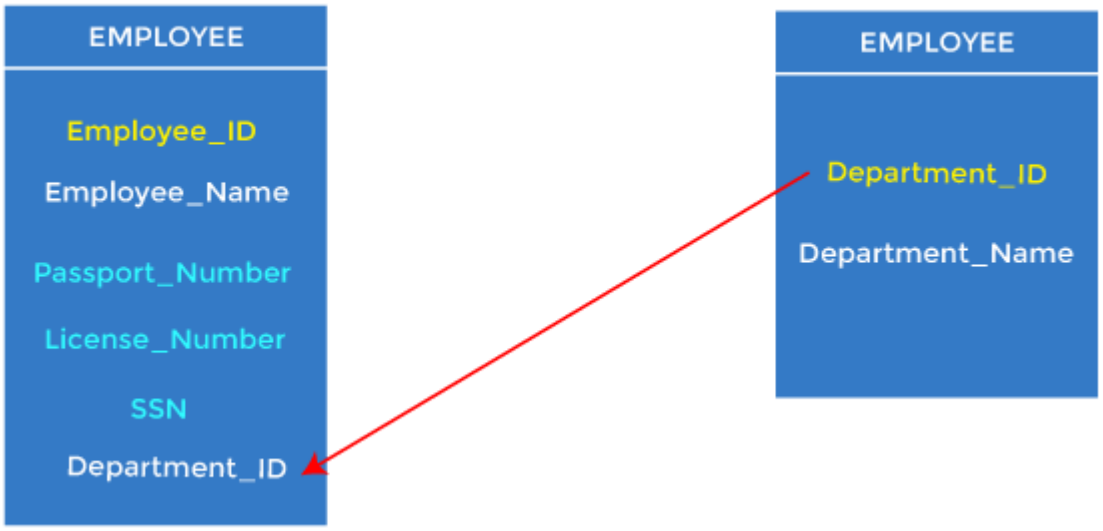


For example: In the above EMPLOYEE table, for(EMPLOEE_ID, EMPLOYEE_NAME), the name of two employees can be the same, but their EMPLYEE_ID can't be the same. Hence, this combination can also be a key.

The super key would be EMPLOYEE-ID (EMPLOYEE_ID, EMPLOYEE-NAME), etc.

4. Foreign key

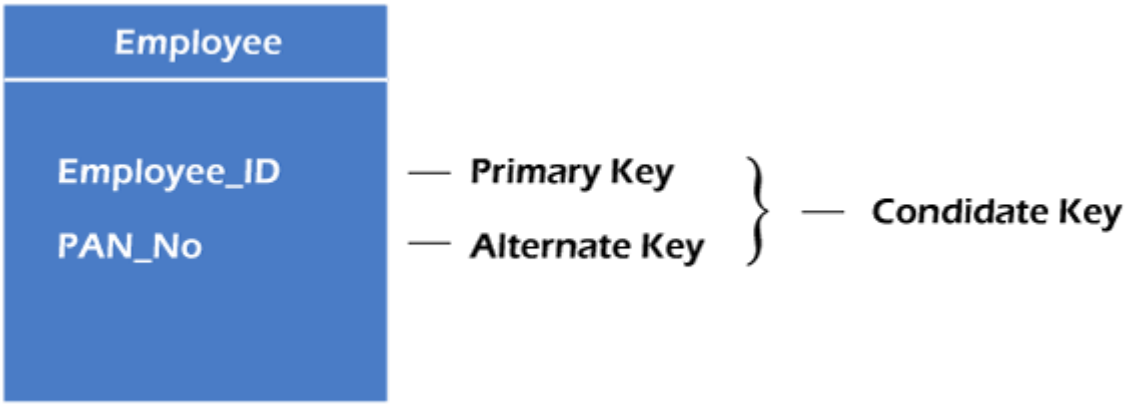
- Foreign keys are the column of the table used to point to the primary key of another table.
- Every employee works in a specific department in a company, and employee and department are two different entities. So we can't store the department's information in the employee table. That's why we link these two tables through the primary key of one table.
- We add the primary key of the DEPARTMENT table, Department_Id, as a new attribute in the EMPLOYEE table.
- In the EMPLOYEE table, Department_Id is the foreign key, and both the tables are related.



5. Alternate key

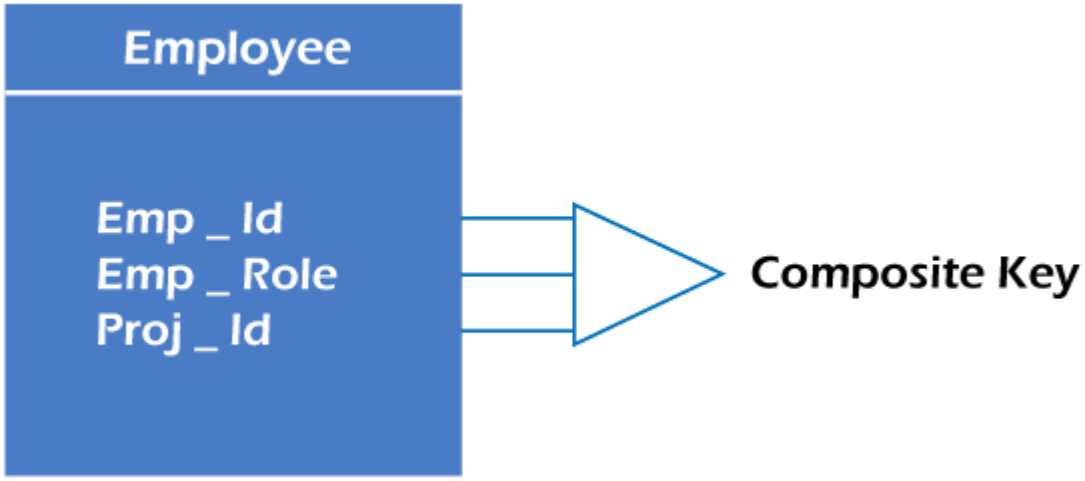
There may be one or more attributes or a combination of attributes that uniquely identify each tuple in a relation. These attributes or combinations of the attributes are called the candidate keys. One key is chosen as the primary key from these candidate keys, and the remaining candidate key, if it exists, is termed the alternate key. **In other words,** the total number of the alternate keys is the total number of candidate keys minus the primary key. The alternate key may or may not exist. If there is only one candidate key in a relation, it does not have an alternate key.

For example, employee relation has two attributes, Employee_Id and PAN_No, that act as candidate keys. In this relation, Employee_Id is chosen as the primary key, so the other candidate key, PAN_No, acts as the Alternate key.

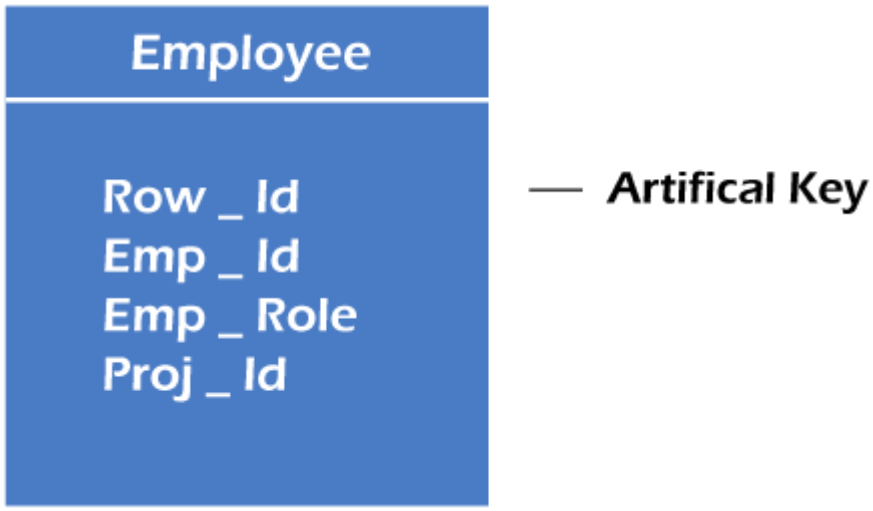


6. Composite key

Whenever a primary key consists of more than one attribute, it is known as a composite key. This key is also known as Concatenated Key.



For example, in employee relations, we assume that an employee may be assigned multiple roles, and an employee may work on multiple projects simultaneously. So the primary key will be composed of all three attributes, namely Emp_ID, Emp_role, and Proj_ID in combination. So these attributes act as a composite key since the primary key comprises more than one attribute.



7. Artificial key

The key created using arbitrarily assigned data are known as artificial keys. These keys are created when a primary key is large and complex and has no relationship with many other relations. The data values of the artificial keys are usually numbered in a serial order.

For example, the primary key, which is composed of Emp_ID, Emp_role, and Proj_ID, is large in employee relations. So it would be better to add a new virtual attribute to identify each tuple in the relation uniquely.

First Normal Form (1NF)

- A relation will be 1NF if it contains an atomic value.
- It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.
- First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

- In the 2NF, relational must be in 1NF.

- In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Math	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English

83	Math
83	Computer

Third Normal Form (3NF)

- A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.
- 3NF is used to reduce the data duplication. It is also used to achieve the data integrity.
- If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency X → Y.

1. X is a super key.
2. Y is a prime attribute, i.e., each element of Y is part of some candidate key.

   Example:

   EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston
444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

   Super key in the table above:

   1. {EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP}....so o n

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd normal form (BCNF)

- BCNF is the advance version of 3NF. It is stricter than 3NF.

- A table is in BCNF if every functional dependency X → Y, X is the super key of the table.
- For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO

Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

1. EMP_ID → EMP_COUNTRY
2. EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID
For the second table: EMP_DEPT
For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

lossless join decomposition

Lossless-join decomposition is a process in which a relation is decomposed into two or more relations. This property guarantees that the extra or less tuple generation problem does not occur and no information is lost from the original relation during the decomposition. It is also known as non-additive join decomposition.

When the sub relations combine again then the new relation must be the same as the original relation was before decomposition.

Consider a relation R if we decomposed it into sub-parts relation R1 and relation R2.

The decomposition is lossless when it satisfies the following statement −

- If we union the sub Relation R1 and R2 then it must contain all the attributes that are available in the original relation R before decomposition.
- Intersections of R1 and R2 cannot be Null. The sub relation must contain a common attribute. The common attribute must contain unique data.

The common attribute must be a super key of sub relations either R1 or R2.

Here,

R = (A, B, C)

R1 = (A, B)

R2 = (B, C)

The relation R has three attributes A, B, and C. The relation R is decomposed into two relation R1 and R2. . R1 and R2 both have 2-2 attributes.The common attributes are B.

The Value in Column B must be unique. if it contains a duplicate value then the Lossless-join decomposition is not possible.

Draw a table of Relation R with Raw Data −

R (A, B, C)

A	B	C
12	25	34
10	36	09
12	42	30

It decomposes into the two sub relations −

R1 (A, B)

A	B
12	25
10	36
12	42

R2 (B, C)

B	C
25	34
36	09
42	30

Now, we can check the first condition for Lossless-join decomposition.

The union of sub relation R1 and R2 is the same as relation R.

$R_1 U R_2 = R$

We get the following result −

A	B	C
12	25	34
10	36	09
12	42	30

The relation is the same as the original relation R. Hence, the above decomposition is Lossless-join decomposition.

Advantages of Lossless Join and Dependency Preserving Decomposition:

Improved Data Integrity: Lossless join and dependency preserving decomposition help to maintain the data integrity of the original relation by ensuring that all dependencies are preserved.
Reduced Data Redundancy: These techniques help to reduce data redundancy by breaking down a relation into smaller, more manageable relations.
Improved Query Performance: By breaking down a relation into smaller, more focused relations, query performance can be improved.
Easier Maintenance and Updates: The smaller, more focused relations are easier to maintain and update than the original relation, making it easier to modify the database schema and update the data.
Better Flexibility: Lossless join and dependency preserving decomposition can improve the flexibility of the database system by allowing for easier modification of the schema.

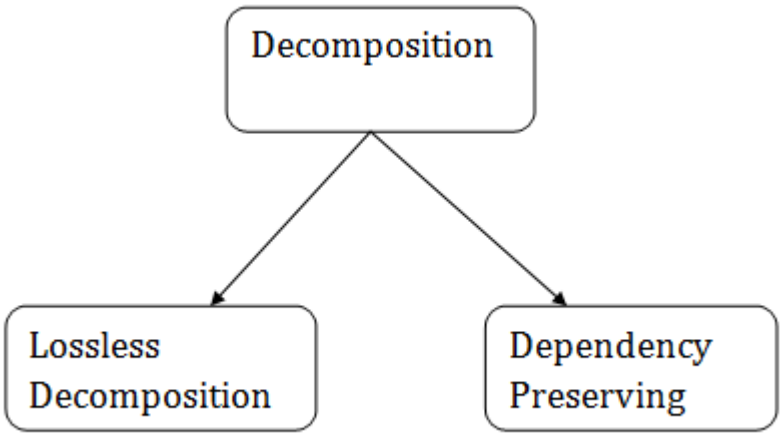
Disadvantages of Lossless Join and Dependency Preserving Decomposition:

Increased Complexity: Lossless join and dependency preserving decomposition can increase the complexity of the database system, making it harder to understand and manage.
Costly: Decomposing relations can be costly, especially if the database is large and complex. This can require additional resources, such as hardware and personnel.
Reduced Performance: Although query performance can be improved in some cases, in others, lossless join and dependency preserving decomposition can result in reduced query performance due to the need for additional join operations.
Limited Scalability: These techniques may not scale well in larger databases, as the number of smaller, focused relations can become unwieldy.

Relational Decomposition

- When a relation in the relational model is not in appropriate normal form then the decomposition of a relation is required.
- In a database, it breaks the table into multiple tables.
- If the relation has no proper decomposition, then it may lead to problems like loss of information.
- Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy.

Types of Decomposition



Lossless Decomposition

- If the information is not lost from the relation that is decomposed, then the decomposition will be lossless.
- The lossless decomposition guarantees that the join of relations will result in the same relation as it was decomposed.
- The relation is said to be lossless decomposition if natural joins of all the decomposition give the original relation.

Example:

EMPLOYEE_DEPARTMENT table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales

33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

The above relation is decomposed into two relations EMPLOYEE and DEPARTMENT

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY
22	Denim	28	Mumbai
33	Alina	25	Delhi
46	Stephan	30	Bangalore
52	Katherine	36	Mumbai
60	Jack	40	Noida

DEPARTMENT table

DEPT_ID	EMP_ID	DEPT_NAME
827	22	Sales
438	33	Marketing
869	46	Finance
575	52	Production

678	60	Testing

Now, when these two relations are joined on the common column "EMP_ID", then the resultant relation will look like:

Employee ⋈ Department

EMP_ID	EMP_NAME	EMP_AGE	EMP_CITY	DEPT_ID	DEPT_NAME
22	Denim	28	Mumbai	827	Sales
33	Alina	25	Delhi	438	Marketing
46	Stephan	30	Bangalore	869	Finance
52	Katherine	36	Mumbai	575	Production
60	Jack	40	Noida	678	Testing

Hence, the decomposition is Lossless join decomposition.

Dependency Preserving

- It is an important constraint of the database.
- 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.
- For example, suppose there is a relation R (A, B, C, D) with functional dependency set (A->BC). The relational R is decomposed into R1(ABC) and R2(AD) which is dependency preserving because FD A->BC is a part of relation R1(ABC).

Fourth normal form (4NF)

- 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.

Example

STUDENT

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_COURSE

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

STUDENT_HOBBY

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

Fifth normal form (5NF)

- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.
- 5NF is satisfied when all the tables are broken into as many tables as possible in order to avoid redundancy.
- 5NF is also known as Project-join normal form (PJ/NF).

Example

SUBJECT	LECTURER	SEMESTER
Computer	Anshika	Semester 1
Computer	John	Semester 1
Math	John	Semester 1
Math	Akash	Semester 2
Chemistry	Praveen	Semester 1

In the above table, John takes both Computer and Math class for Semester 1 but he doesn't take Math class for Semester 2. In this case, combination of all these fields required to identify a valid data.

Suppose we add a new Semester as Semester 3 but do not know about the subject and who will be taking that subject so we leave Lecturer and Subject as NULL. But all three columns together acts as a primary key, so we can't leave other two columns blank.

So to make the above table into 5NF, we can decompose it into three relations P1, P2 & P3:

P1

SEMESTER	SUBJECT
Semester 1	Computer

Semester 1	Math
Semester 1	Chemistry
Semester 2	Math

P2

SUBJECT	LECTURER
Computer	Anshika
Computer	John
Math	John
Math	Akash
Chemistry	Praveen

P3

SEMSTER	LECTURER
Semester 1	Anshika
Semester 1	John
Semester 1	John
Semester 2	Akash
Semester 1	Praveen

Domain-Key Normal Form

A relation is in DKNF when insertion or delete anomalies are not present in the database. Domain-Key Normal Form is the highest form of Normalization. The reason is that the insertion and updation anomalies are removed. The constraints are verified by the domain and key constraints.

A table is in Domain-Key normal form only if it is in 4NF, 3NF and other normal forms. It is based on constraints −

Domain Constraint

Values of an attribute had some set of values, for example, EmployeeID should be four digits long −

EmpID	EmpName	EmpAge
0921	Tom	33
0922	Jack	31

Key Constraint

An attribute or its combination is a candidate key

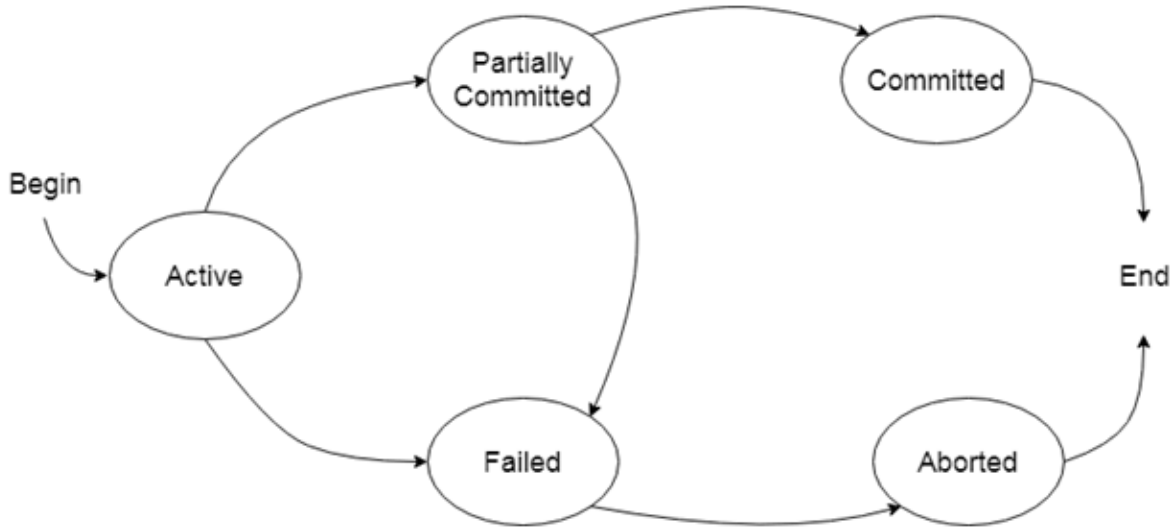
General Constraint

Predicate on the set of all relations.

Every constraint should be a logical sequence of the domain constraints and key constraints applied to the relation. The practical utility of DKNF is less.

States of Transaction

In a database, the transaction can be in one of the following states -



Active state

- The active state is the first state of every transaction. In this state, the transaction is being executed.
- For example: Insertion or deletion or updating a record is done here. But all the records are still not saved to the database.

Partially committed

- In the partially committed state, a transaction executes its final operation, but the data is still not saved to the database.
- In the total mark calculation example, a final display of the total marks step is executed in this state.

Committed

A transaction is said to be in a committed state if it executes all its operations successfully. In this state, all the effects are now permanently saved on the database system.

Failed state

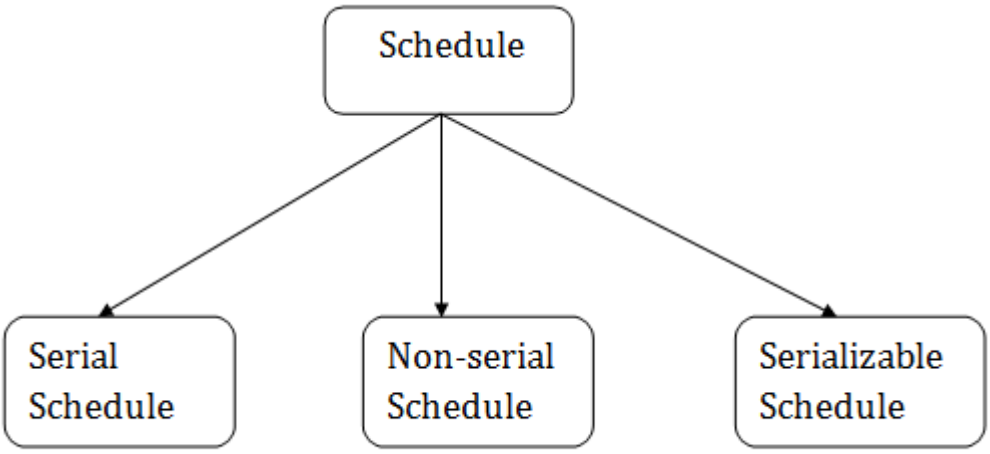
- If any of the checks made by the database recovery system fails, then the transaction is said to be in the failed state.
- In the example of total mark calculation, if the database is not able to fire a query to fetch the marks, then the transaction will fail to execute.

Aborted

- If any of the checks fail and the transaction has reached a failed state then the database recovery system will make sure that the database is in its previous consistent state. If not then it will abort or roll back the transaction to bring the database into a consistent state.
- If the transaction fails in the middle of the transaction then before executing the transaction, all the executed transactions are rolled back to its consistent state.
- After aborting the transaction, the database recovery module will select one of the two operations:
    1. Re-start the transaction
    2. Kill the transaction

Schedule

A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



1. Serial Schedule

The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.

For example: Suppose there are two transactions T1 and T2 which have some operations. If it has no interleaving of operations, then there are the following two possible outcomes:

1. Execute all the operations of T1 which was followed by all the operations of T2.
2. Execute all the operations of T1 which was followed by all the operations of T2.

- In the given (a) figure, Schedule A shows the serial schedule where T1 followed by T2.
- In the given (b) figure, Schedule B shows the serial schedule where T2 followed by T1.

2. Non-serial Schedule

- If interleaving of operations is allowed, then there will be non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

3. Serializable schedule

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have interleaving of their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

(a)

Time

T1	T2

read(A);
A := A – N;
write(A);
read(B);
B := B + N;
write(B);
	read(A);
	A := A + M;
	write(A);

Schedule A

(b)

Time

T1	T2

read(A);
A:= A - N;
write(A);
read(B);
B:= B +N;
write(B);
	read(A);
	A := A + M;
	write(A);

Schedule B

(c)

Time

T1	T2
read(A); A := A – N;	
	read(A); A := A + M;
write(A); read(B);	
	write(A);
B := B + N; write(B);	

Schedule C

(d)

Time

T1	T2

read(A);
A := A – N;
write(A);	

	read(A);
	A := A + M;
	write(A);

read(B);
B := B + N;
write(B);	

Schedule D

Here,

Schedule A and Schedule B are serial schedule.

Schedule C and Schedule D are Non-serial schedule.

Serializability in DBMS

The backbone of the most modern application is the form of DBMS. When we design the form properly, then it provides high-performance and relative storage solutions to our application. In this topic, we are going to explain the serializability concept and how this concept affects the DBMS deeply. We also understand the concept of serializability with some examples. Finally, we will conclude this topic with an example of the importance of serializability.

What is Serializability in DBMS?

In the field of computer science, serializability is a term that is a property of the system that describes how the different process operates the shared data. If the result given by the system is similar to the operation performed by the system, then in this situation, we call that system serializable. Here the cooperation of the system means there is no overlapping in the execution of the data. In DBMS, when the data is being written or read then, the DBMS can stop all the other processes from accessing the data.

In the MongoDB developer certificate, the DBMS uses various locking systems to allow the other processes while maintaining the integrity of the data. In MongoDB, the most restricted level for serializability is the employee can be restricted by two-phase locking or 2PL. In the first phase of the locking level, the data objects are locked before the execution of the operation. When the transaction has been accomplished, then the lock for the data object is released. This process guarantees that there is no conflict in operation and that all the transaction views the database as a conflict database.

The two-phase locking or 2PL system provides a strong guarantee for the conflict of the database.

It can reduce the decreased performance and then increase the overhead acquiring capacity and then release the lock of the data. As a result, the system allows the constraint serializability for better performance of the DBMS. This ensures that the final result is the same as some sequential execution and performs the improvement of the operation that is involved in the database.

Thus, serializability is the system's property that describes how the different process operates the shared data. In DBMS, the overall Serializable property is adopted by locking the data during the execution of other processes. Also, serializability ensures that the final result is equivalent to the sequential operation of the data.

What is a Serializable Schedule?

- In DBMS, the Serializable schedule is a property in which the read and write operation sequence does not disturb the serializability property. This property ensures that the

transaction is executed automatically with the other transaction. In DBMS, the order of the serializability must be the same as some serial schedules of the same transaction.

- In DBMS, there are several algorithms available to check the serializability of the database. One of the most important algorithms is the conflict serializability algorithm. The conflict serializability algorithm is the ability to check the potential of the conflict in the database. When the two transactions access the same data, this conflict occurs in the database. If there is no conflict, then there is guaranteed serializability in the database. However, if there is a conflict occurs, then there is a chance of serializability.
- Another algorithm that is used to check the serializability of the database is the DBMS algorithm. With the help of the DBMS algorithm, we can check the potential mutual dependencies between two transactions. When the two transactions give the correct output, then mutual dependencies exist. When there are no mutual dependencies, then there is a guaranteed serializable in the database. However, if there are mutual dependencies, then there will be a chance of serializable.
- We can also check the serializability of the database by using the precedence graph algorithm. A precedence relationship exists when one transaction must precede another transaction for the schedule to be valid. If there are no cycles, then the serializability of schedules in DBMS is guaranteed. However, if there are cycles, the schedule may or may not be serializable. To understand different algorithms comprehensively, take the MongoDB Administration certification and get expert analysis on the concept of serializability in DBMS.

Types of Serializability

In DBMS, all the transaction should be arranged in a particular order, even if all the transaction is concurrent. If all the transaction is not serializable, then it produces the incorrect result.

In DBMS, there are different types of serializable. Each type of serializable has some advantages and disadvantages. The two most common types of serializable are view serializability and conflict serializability.

1. Conflict Serializability

Conflict serializability is a type of conflict operation in serializability that operates the same data item that should be executed in a particular order and maintains the consistency of the database. In DBMS, each transaction has some unique value, and every transaction of the database is based on that unique value of the database.

This unique value ensures that no two operations having the same conflict value are executed concurrently. For example, let's consider two examples, i.e., the order table

and the customer table. One customer can have multiple orders, but each order only belongs to one customer. There is some condition for the conflict serializability of the database. These are as below.

- Both operations should have different transactions.
- Both transactions should have the same data item.
- There should be at least one write operation between the two operations.

If there are two transactions that are executed concurrently, one operation has to add the transaction of the first customer, and another operation has added by the second operation. This process ensures that there would be no inconsistency in the database.

2. View Serializability

View serializability is a type of operation in the serializable in which each transaction should produce some result and these results are the output of proper sequential execution of the data item. Unlike conflict serialized, the view serializability focuses on preventing inconsistency in the database. In DBMS, the view serializability provides the user to view the database in a conflicting way.

In DBMS, we should understand schedules S1 and S2 to understand view serializability better. These two schedules should be created with the help of two transactions T1 and T2. To maintain the equivalent of the transaction each schedule has to obey the three transactions. These three conditions are as follows.

- The first condition is each schedule has the same type of transaction. The meaning of this condition is that both schedules S1 and S2 must not have the same type of set of transactions. If one schedule has committed the transaction but does not match the transaction of another schedule, then the schedule is not equivalent to each other.
- The second condition is that both schedules should not have the same type of read or write operation. On the other hand, if schedule S1 has two write operations while schedule S2 has one write operation, we say that both schedules are not equivalent to each other. We may also say that there is no problem if the number of the read operation is different, but there must be the same number of the write operation in both schedules.
- The final and last condition is that both schedules should not have the same conflict. Order of execution of the same data item. For example, suppose the transaction of schedule S1 is T1, and the transaction of schedule S2 is T2. The transaction T1 writes the data item A, and the transaction T2 also writes the data item A. in this case, the schedule is not equivalent to each other. But if the schedule has the same number of

each write operation in the data item then we called the schedule equivalent to each other.

Testing of Serializability in DBMS with Examples

Serializability is a type of property of DBMS in which each transaction is executed independently and automatically, even though these transactions are executed concurrently. In other words, we can say that if there are several transactions executed concurrently, then the main work of the serializability function is to arrange these several transactions in a sequential manner.

For better understanding, let's explain these with an example. Suppose there are two users Sona and Archita. Each executes two transactions. Let's transactions T1 and T2 are executed by Sona, and T3 and T4 are executed by Archita. Suppose transaction T1 reads and writes the data item A, transaction T2 reads the data item B, transaction T3 reads and writes the data item C and transaction T4 reads the data item D. Lets the schedule the above transaction as below.

1. •
2. T1: Read A → Write A→ Read B → Write B`
3. •
4. `T2: Read B → Write B`
5. •
6. `T3: Read C → Write C→ Read D → Write D`
7. • `T4: Read D → Write D

Let's first discuss why these transactions are not serializable.

In order for a schedule to be considered serializable, it must first satisfy the conflict serializability property. In our example schedule above, notice that Transaction 1 (T1) and Transaction 2 (T2) read data item B before either writing it. This causes a conflict between T1 and T2 because they are both trying to read and write the same data item concurrently. Therefore, the given schedule does not conflict with serializability.

However, there is another type of serializability called view serializability which our example does satisfy. View serializability requires that if two transactions cannot see each other's updates (i.e., one transaction cannot see the effects of another concurrent transaction), the schedule is considered to view serializable. In our example, Transaction 2 (T2) cannot see any updates made by Transaction 4 (T4) because they do not share common data items. Therefore, the schedule is viewed as serializable.

It's important to note that conflict serializability is a stronger property than view serializability because it requires that all potential conflicts be resolved before any updates are made (i.e., each transaction must either read or write each data item before

any other transaction can write it). View serializability only requires that if two transactions cannot see each other's updates, then the schedule is view serializable & it doesn't matter whether or not there are potential conflicts between them.

All in all, both properties are necessary for ensuring correctness in concurrent transactions in a database management system.

Benefits of Serializability in DBMS

Below are the benefits of using the serializable in the database.

1. **Predictable execution:** In serializable, all the threads of the DBMS are executed at one time. There are no such surprises in the DBMS. In DBMS, all the variables are updated as expected, and there is no data loss or corruption.
2. **Easier to Reason about & Debug:** In DBMS all the threads are executed alone, so it is very easier to know about each thread of the database. This can make the debugging process very easy. So we don't have to worry about the concurrent process.
3. **Reduced Costs:** With the help of serializable property, we can reduce the cost of the hardware that is being used for the smooth operation of the database. It can also reduce the development cost of the software.
4. **Increased Performance:**In some cases, serializable executions can perform better than their non-serializable counterparts since they allow the developer to optimize their code for performance.

Conclusion

DBMS transactions must follow the ACID properties to be considered serializable. There are different types of serializability in DBMS, each with its own benefits and drawbacks. In most cases, selecting the right type of serializability will come down to a trade-off between performance and correctness.

Selecting the wrong type of serializability can lead to errors in your database that can be difficult to debug and fix. Hopefully, this guide has given you a better understanding of how Serializability in DBMS works and what kinds of serializability exist.

DBMS Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent*.

For example:

Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction T_X writes the value of account A that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

For example:

Consider two transactions T_X and T_Y in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300.
- At time t2, transaction T_X adds $50 to account A that becomes $350.
- At time t3, transaction T_X writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_Y reads account A that will be read as $350.
- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_X and T_Y, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_X reads the value from account A, i.e., $300.
- At time t2, transaction T_Y reads the value from account A, i.e., $300.
- At time t3, transaction T_Y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_Y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

DBMS Concurrency Control

Concurrency Control is the management procedure that is required for controlling concurrent execution of the operations that take place on a database.

But before knowing about concurrency control, we should know about concurrent execution.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.
- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

In a database transaction, the two main operations are **READ** and **WRITE** operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and

the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs *when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.*

For example:

Consider the below diagram where two transactions T_X and T_Y, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300 (only read).
- At time t2, transaction T_X deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction T_Y reads the value of account A that will be $300 only because T_X didn't update the value yet.
- At time t4, transaction T_Y adds $100 to account A that becomes $400 (only added but not updated/write).

- At time t6, transaction T_X writes the value of account A that will be updated as $250 only, as T_Y didn't update the value yet.
- Similarly, at time t7, transaction T_Y writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by T_X is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Dirty Read Problems (W-R Conflict)

The dirty read problem occurs *when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction. There comes the Read-Write Conflict between both transactions.*

For example:

Consider two transactions T_X and T_Y in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_X	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction T_X reads the value of account A, i.e., $300.
- At time t2, transaction T_X adds $50 to account A that becomes $350.
- At time t3, transaction T_X writes the updated value in account A, i.e., $350.
- Then at time t4, transaction T_Y reads account A that will be read as $350.

- Then at time t5, transaction T_X rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction T_Y as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example:

Consider two transactions, T_X and T_Y, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction T_X reads the value from account A, i.e., $300.
- At time t2, transaction T_Y reads the value from account A, i.e., $300.
- At time t3, transaction T_Y updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction T_Y writes the updated value, i.e., $400.
- After that, at time t5, transaction T_X reads the available value of account A, and that will be read as $400.
- It means that within the same transaction T_X, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction T_Y, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Concurrency Control

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

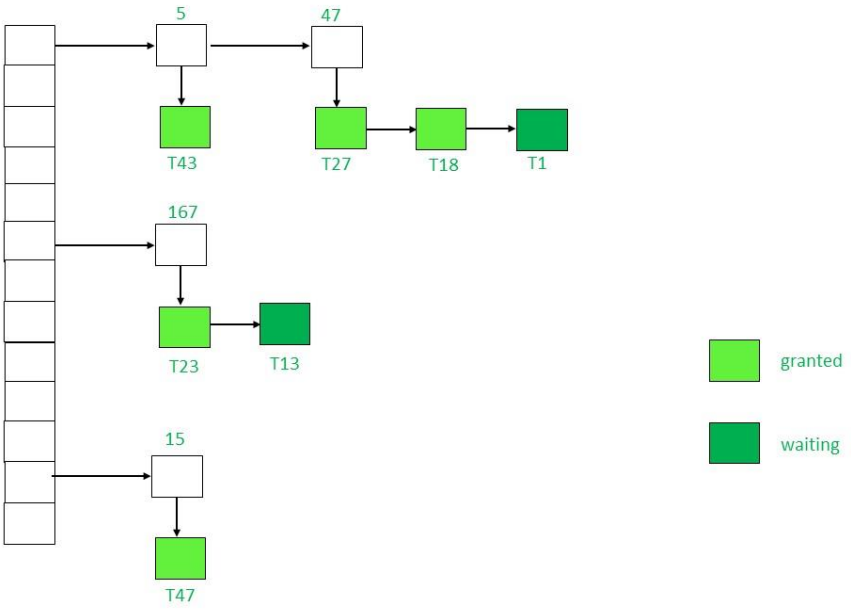
- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

Locking in DBMS

Locking protocols are used in database management systems as a means of concurrency control. Multiple transactions may request a lock on a data item simultaneously. Hence, we require a mechanism to manage the locking requests made by transactions. Such a mechanism is called as **Lock Manager**. It relies on the process of message passing where transactions and lock manager exchange messages to handle the locking and unlocking of data items. **Data structure used in Lock Manager** – The data structure required for implementation of locking is called as **Lock table**.

1. It is a hash table where name of data items are used as hashing index.
2. Each locked data item has a linked list associated with it.
3. Every node in the linked list represents the transaction which requested for lock, mode of lock requested (mutual/exclusive) and current status of the request (granted/waiting).
4. Every new lock request for the data item will be added in the end of linked list as a new node.
5. Collisions in hash table are handled by technique of separate chaining.

Consider the following example of lock table:



Explanation: In the above figure, the locked data items present in lock table are 5, 47, 167 and 15. The transactions which have requested for lock have been represented by a linked list shown below them using a downward arrow. Each node in linked list has the name of transaction which has requested the data item like T33, T1, T27 etc. The colour of node represents the status i.e. whether lock has been granted or waiting. Note that a collision has occurred for data item 5 and 47. It has been resolved by separate chaining where each data item belongs to a linked list. The data item is acting as header for linked list containing the locking request. **Working of Lock Manager –**

1. Initially the lock table is empty as no data item is locked.
2. Whenever lock manager receives a lock request from a transaction T_i on a particular data item Q_i following cases may arise:
    - If Q_i is not already locked, a linked list will be created and lock will be granted to the requesting transaction T_i.
    - If the data item is already locked, a new node will be added at the end of its linked list containing the information about request made by T_i.
3. If the lock mode requested by T_i is compatible with lock mode of transaction currently having the lock, T_i will acquire the lock too and status will be changed to ‘granted’. Else, status of T_i’s lock will be ‘waiting’.
4. If a transaction T_i wants to unlock the data item it is currently holding, it will send an unlock request to the lock manager. The lock manager will delete T_i’s node from this linked list. Lock will be granted to the next transaction in the list.
5. Sometimes transaction T_i may have to be aborted. In such a case all the waiting request made by T_i will be deleted from the linked lists

present in lock table. Once abortion is complete, locks held by T_i will also be released.

Advantages:

Data Consistency: Locking can help ensure data consistency by preventing multiple users from modifying the same data simultaneously. By controlling access to shared resources, locking can help prevent data conflicts and ensure that the database remains in a consistent state.
Isolation: Locking can ensure that transactions are executed in isolation from other transactions, preventing interference between transactions and reducing the risk of data inconsistencies.
Granularity: Locking can be implemented at different levels of granularity, allowing for more precise control over shared resources. For example, row-level locking can be used to lock individual rows in a table, while table-level locking can be used to lock entire tables.
Availability: Locking can help ensure the availability of shared resources by preventing users from monopolizing resources or causing resource starvation.

Disadvantages:

Overhead: Locking requires additional overhead, such as acquiring and releasing locks on shared resources. This overhead can lead to slower performance and increased resource consumption, particularly in systems with high levels of concurrency.
Deadlocks: Deadlocks can occur when two or more transactions are waiting for each other to release resources, causing a circular dependency that can prevent any of the transactions from completing. Deadlocks can be difficult to detect and resolve, and can result in reduced throughput and increased latency.
Reduced Concurrency: Locking can limit the number of users or applications that can access the database simultaneously. This can lead to reduced concurrency and slower performance in systems with high levels of concurrency.
Complexity: Implementing locking can be complex, particularly in distributed systems or in systems with complex transactional logic. This complexity can lead to increased development and maintenance costs.

Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.
- Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.
- The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Recoverability of Schedule

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		
Failure Point				
Commit;				

The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it

reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

Irrecoverable schedule: The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
Failure Point				
Commit;				
		Commit;		

The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.

T1	T1's buffer space	T2	T2's buffer space	Database
				A = 6500
Read(A);	A = 6500			A = 6500
A = A - 500;	A = 6000			A = 6500
Write(A);	A = 6000			A = 6000
Commit;		Read(A);	A = 6000	A = 6000
		A =A + 1000;	A = 7000	A = 6000
		Write(A);	A = 7000	A = 7000
		Commit;		

The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

Multiple Granularity

Let's start by understanding the meaning of granularity.

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

- It can be defined as hierarchically breaking up the database into blocks which can be locked.
- The Multiple Granularity protocol enhances concurrency and reduces lock overhead.
- It maintains the track of what to lock and how to lock.
- It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area. The higher level database consists of exactly these areas.
- The area consists of children nodes which are known as files. No file can be present in more than one area.
- Finally, each file contains child nodes known as records. The file has exactly those records that are its child nodes. No records represent in more than one file.
- Hence, the levels of the tree starting from the top level are as follows:
    1. Database
    2. Area
    3. File
    4. Record

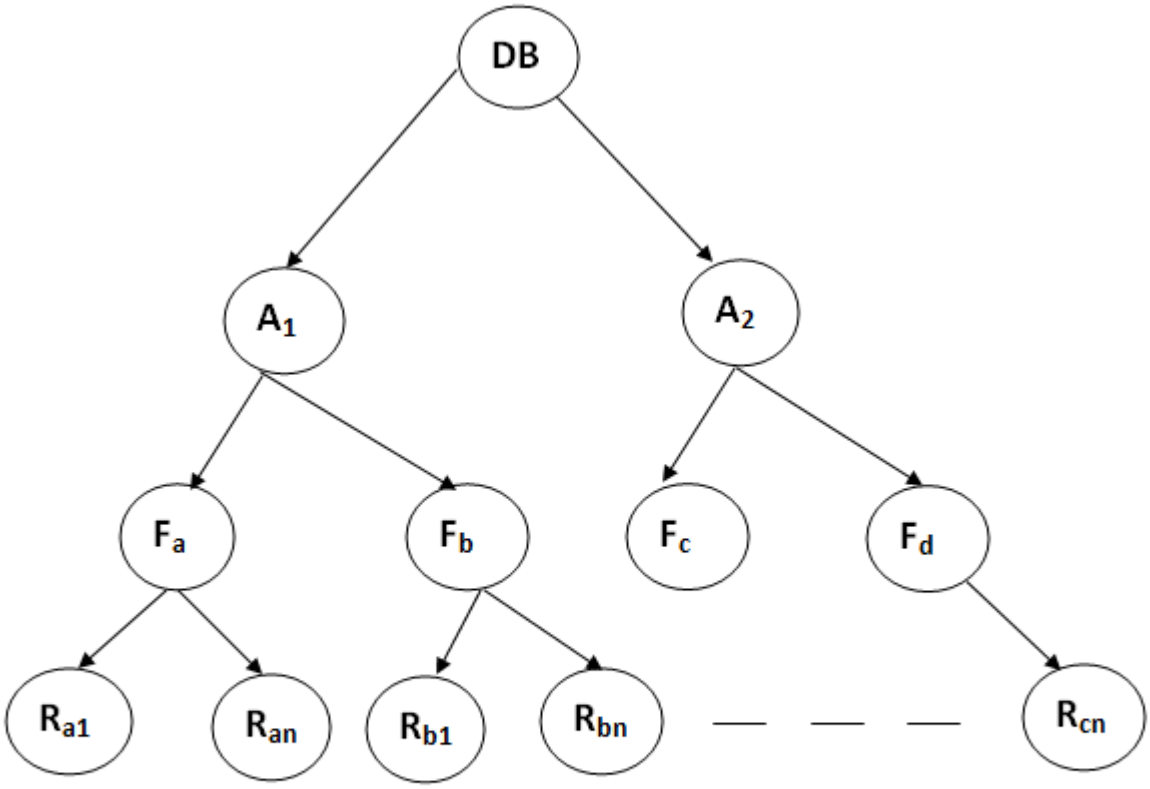


Figure: Multi Granularity tree Hierarchy

Deadlock Detection And Recovery

Deadlock detection and recovery is the process of detecting and resolving deadlocks in an operating system. A deadlock occurs when two or more processes are blocked, waiting for each other to release the resources they need. This can lead to a system-wide stall, where no process can make progress.

There are two main approaches to deadlock detection and recovery:

1. **Prevention**: The operating system takes steps to prevent deadlocks from occurring by ensuring that the system is always in a safe state, where deadlocks cannot occur. This is achieved through resource allocation algorithms such as the Banker's Algorithm.
2. **Detection and Recovery:** If deadlocks do occur, the operating system must detect and resolve them. Deadlock detection algorithms, such as the Wait-For Graph, are used to identify deadlocks, and recovery algorithms, such as the Rollback and Abort algorithm, are used to resolve them. The recovery algorithm releases the resources held by one or more processes, allowing the system to continue to make progress.

Difference Between Prevention and Detection/Recovery: Prevention aims to avoid deadlocks altogether by carefully managing resource allocation, while

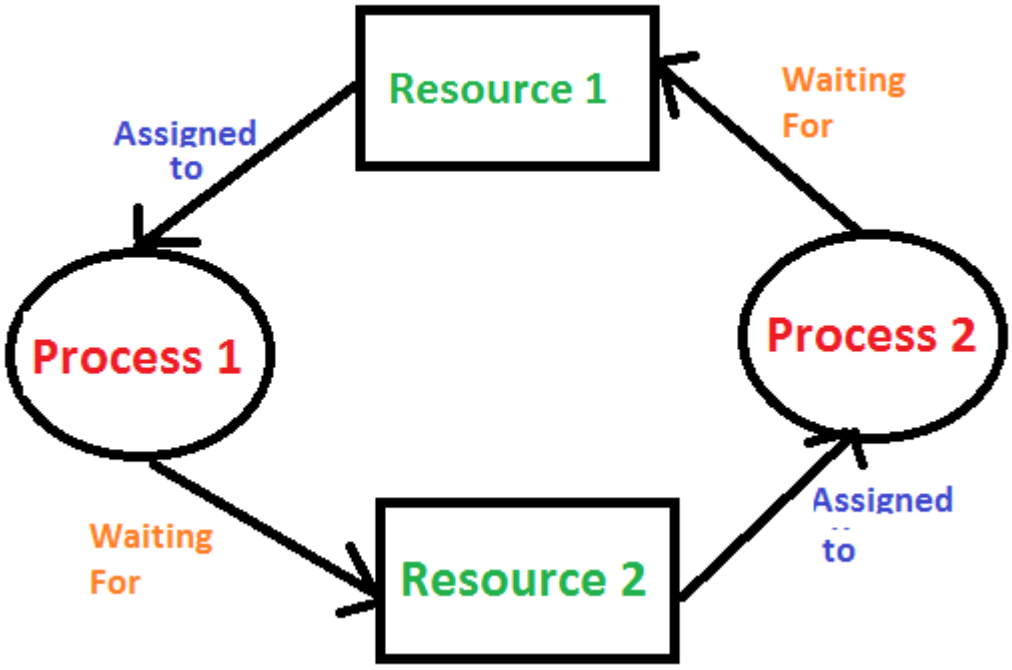
detection and recovery aim to identify and resolve deadlocks that have already occurred.
Deadlock detection and recovery is an important aspect of operating system design and management, as it affects the stability and performance of the system. The choice of deadlock detection and recovery approach depends on the specific requirements of the system and the trade-offs between performance, complexity, and risk tolerance. The operating system must balance these factors to ensure that deadlocks are effectively detected and resolved.

In the previous post, we discussed Deadlock Prevention and Avoidance. In this post, the Deadlock Detection and Recovery technique to handle deadlock is discussed.

Deadlock Detection :

1. If resources have a single instance –

In this case for Deadlock detection, we can run an algorithm to check for the cycle in the Resource Allocation Graph. The presence of a cycle in the graph is a sufficient condition for deadlock.



In the above diagram, resource 1 and resource 2 have single instances. There is a cycle R1 → P1 → R2 → P2. So, Deadlock is Confirmed.

2. If there are multiple instances of resources –

Detection of the cycle is necessary but not a sufficient condition for deadlock detection, in this case, the system may or may not be in deadlock varies according to different situations.

3. Wait-For Graph Algorithm –

The Wait-For Graph Algorithm is a deadlock detection algorithm used to detect deadlocks in a system where resources can have multiple instances. The algorithm

works by constructing a Wait-For Graph, which is a directed graph that represents the dependencies between processes and resources.

Deadlock Recovery :
A traditional operating system such as Windows doesn't deal with deadlock recovery as it is a time and space-consuming process. Real-time operating systems use Deadlock recovery.

1. **Killing the process –**
   Killing all the processes involved in the deadlock. Killing process one by one. After killing each process check for deadlock again and keep repeating the process till the system recovers from deadlock. Killing all the processes one by one helps a system to break circular wait conditions.
2. **Resource Preemption –**
   Resources are preempted from the processes involved in the deadlock, and preempted resources are allocated to other processes so that there is a possibility of recovering the system from the deadlock. In this case, the system goes into starvation.
3. **Concurrency Control –** Concurrency control mechanisms are used to prevent data inconsistencies in systems with multiple concurrent processes. These mechanisms ensure that concurrent processes do not access the same data at the same time, which can lead to inconsistencies and errors. Deadlocks can occur in concurrent systems when two or more processes are blocked, waiting for each other to release the resources they need. This can result in a system-wide stall, where no process can make progress. Concurrency control mechanisms can help prevent deadlocks by managing access to shared resources and ensuring that concurrent processes do not interfere with each other.

ADVANTAGES OR DISADVANTAGES:

Advantages of Deadlock Detection and Recovery in Operating Systems:

1. **Improved System Stability:** Deadlocks can cause system-wide stalls, and detecting and resolving deadlocks can help to improve the stability of the system.
2. **Better Resource Utilization:** By detecting and resolving deadlocks, the operating system can ensure that resources are efficiently utilized and that the system remains responsive to user requests.
3. **Better System Design**: Deadlock detection and recovery algorithms can provide insight into the behavior of the system and the relationships between processes and resources, helping to inform and improve the design of the system.

Disadvantages of Deadlock Detection and Recovery in Operating Systems:

1. **Performance Overhead:** Deadlock detection and recovery algorithms can introduce a significant overhead in terms of performance, as the system

must regularly check for deadlocks and take appropriate action to resolve them.
2. **Complexity:** Deadlock detection and recovery algorithms can be complex to implement, especially if they use advanced techniques such as the Resource Allocation Graph or Timestamping.
3. **False Positives and Negatives:** Deadlock detection algorithms are not perfect and may produce false positives or negatives, indicating the presence of deadlocks when they do not exist or failing to detect deadlocks that do exist.
4. **Risk of Data Loss**: In some cases, recovery algorithms may require rolling back the state of one or more processes, leading to data loss or corruption.

Database Recovery Techniques in DBMS

Database recovery techniques are used in database management systems (DBMS) to restore a database to a consistent state after a failure or error has occurred. The main goal of recovery techniques is to ensure data integrity and consistency and prevent data loss. There are mainly two types of recovery techniques used in DBMS:

Rollback/Undo Recovery Technique: The rollback/undo recovery technique is based on the principle of backing out or undoing the effects of a transaction that has not completed successfully due to a system failure or error. This technique is accomplished by undoing the changes made by the transaction using the log records stored in the transaction log. The transaction log contains a record of all the transactions that have been performed on the database. The system uses the log records to undo the changes made by the failed transaction and restore the database to its previous state.
Commit/Redo Recovery Technique: The commit/redo recovery technique is based on the principle of reapplying the changes made by a transaction that has been completed successfully to the database. This technique is accomplished by using the log records stored in the transaction log to redo the changes made by the transaction that was in progress at the time of the failure or error. The system uses the log records to reapply the changes made by the transaction and restore the database to its most recent consistent state.
In addition to these two techniques, there is also a third technique called checkpoint recovery. Checkpoint recovery is a technique used to reduce the recovery time by periodically saving the state of the database in a checkpoint file. In the event of a failure, the system can use the checkpoint file to restore the database to the most recent consistent state before the failure occurred, rather than going through the entire log to recover the database.

Overall, recovery techniques are essential to ensure data consistency and availability in DBMS, and each technique has its own advantages and limitations that must be considered in the design of a recovery system

Database systems, like any other computer system, are subject to failures but the data stored in them must be available as and when required. When a database fails it must possess the facilities for fast recovery. It must also have atomicity i.e. either transaction are completed successfully and committed (the effect is recorded permanently in the database) or the transaction should have no effect on the database. There are both automatic and non-automatic ways for both, backing up of data and recovery from any failure situations. The techniques used to recover the lost data due to system crashes, transaction errors, viruses, catastrophic failure, incorrect commands execution, etc. are database recovery techniques. So to prevent data loss recovery techniques based on deferred update and immediate update or backing up data can be used. Recovery techniques are heavily dependent upon the existence of a special file known as a **system log**. It contains information about the start and end of each transaction and any updates which occur during the **transaction**. The log keeps track of all transaction operations that affect the values of database items. This information is needed to recover from transaction failure.

- The log is kept on disk start_transaction(T): This log entry records that transaction T starts the execution.
- read_item(T, X): This log entry records that transaction T reads the value of database item X.
- write_item(T, X, old_value, new_value): This log entry records that transaction T changes the value of the database item X from old_value to new_value. The old value is sometimes known as a before an image of X, and the new value is known as an afterimage of X.
- commit(T): This log entry records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- abort(T): This records that transaction T has been aborted.
- checkpoint: Checkpoint is a mechanism where all the previous logs are removed from the system and stored permanently in a storage disk. Checkpoint declares a point before which the DBMS was in a consistent state, and all the transactions were committed.

Stored Procedure

A **stored procedure** is a group of pre-compiled SQL statements (prepared SQL code) that can be reused again and again.

They can be used to perform a wide range of database operations such as inserting, updating, or deleting data, generating reports, and performing complex calculations. Stored procedures are very useful because they allow you to encapsulate (bundle) a set of SQL statements as a single unit and execute them repeatedly with different parameters, making it easy to manage and reuse code.

Procedures have similar structure as functions; they accept parameters and perform operations when we call them; but the difference between them is that SQL stored procedures are simpler to write or create, whereas functions have a more rigid structure and support fewer clauses and functionality.

Syntax

The basic syntax to create a stored procedure in SQL is as follows −

```
CREATE PROCEDURE procedure_name
  @parameter1 datatype,
  @parameter2 datatype
AS
BEGIN
  -- SQL statements to be executed
END
```

Where,

- The CREATE PROCEDURE statement is used to create the procedure. After creating the procedure, we can define any input parameters that the procedure may require. These parameters are preceded by the '@' symbol and followed by their respective data types.
- The AS keyword is used to begin the procedure definition. The SQL statements that make up the procedure are placed between the BEGIN and END keywords.

Creating a Procedure

We can create a stored procedure using the CREATE PROCEDURE statement in SQL. Following are the simple steps for creating a stored procedure −

- Choose a name for the procedure.
- Write the SQL code for the procedure, including to create the procedure in SQL Server.
- We can then test the stored procedure by executing it with different input parameters.

Example

To understand it better let us consider the CUSTOMERS table which contains the personal details of customers including their name, age, address and salary etc. as shown below −

```
CREATE TABLE CUSTOMERS (
   ID INT NOT NULL,
   NAME VARCHAR (20) NOT NULL,
   AGE INT NOT NULL,
   ADDRESS CHAR (25),
   SALARY DECIMAL (18, 2),
   PRIMARY KEY (ID)
);
```

Now insert values into this table using the INSERT statement as follows −

```
INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
```

```
VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (6, 'Komal', 22, 'MP', 4500.00 );


INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
VALUES (7, 'Muffy', 24, 'Indore', 10000.00 );
```

The table will be created as −

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
|  7 | Muffy    |  24 | Indore    | 10000.00 |
+----+----------+-----+-----------+----------+
```

Now, let's look at a simple example of creating a stored procedure that takes an input parameter and returns a result set.

In the following query, we are trying to first create the stored procedure with the name ‘GetCustomerInfo’. We then provide it with a single input parameter called @CutomerAge. The stored procedure then selects all records from the ‘CUSTOMERS’ table where the CutomerAge matches the input parameter.

```
CREATE PROCEDURE GetCustomerInfo

   @CutomerAge INT

AS

BEGIN
```

```
   SELECT * FROM CUSTOMERS

   WHERE AGE = @CutomerAge

END
```

Output

This would produce the following result −

```
Commands completed successfully.
```

Verification

To verify the changes, once we have created the stored procedure, we can test it by executing it with different input parameters as shown in the query below −

```
EXEC GetCustomerInfo @CutomerAge = 25
```

This will return all columns from the CUSTOMERS table where the cutomer's age is 25.

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
+----+----------+-----+-----------+----------+
```

SQL Trigger

Trigger: A trigger is a stored procedure in database which automatically invokes whenever a special event in the database occurs. For example, a trigger can be invoked when a row is inserted into a specified table or when certain table columns are being updated.

Syntax:

```
create trigger [trigger_name]

[before | after]

{insert | update | delete}

on [table_name]

[for each row]

[trigger_body]
```

Explanation of syntax:

1. create trigger [trigger_name]: Creates or replaces an existing trigger with the trigger_name.
2. [before | after]: This specifies when the trigger will be executed.

3. {insert | update | delete}: This specifies the DML operation.
4. on [table_name]: This specifies the name of the table associated with the trigger.
5. [for each row]: This specifies a row-level trigger, i.e., the trigger will be executed for each row being affected.
6. [trigger_body]: This provides the operation to be performed as trigger is fired

BEFORE and AFTER of Trigger:

BEFORE triggers run the trigger action before the triggering statement is run.
AFTER triggers run the trigger action after the triggering statement is run.

Example:

Given Student Report Database, in which student marks assessment is recorded. In such schema, create a trigger so that the total and percentage of specified marks is automatically inserted whenever a record is insert.
Here, as trigger will invoke before record is inserted so, BEFORE Tag can be used.

Suppose the database Schema –

```
mysql> desc Student;
```

```
+-------+-------------+------+-----+---------+----------------+
| Field | Type        | Null | Key | Default | Extra          |
+-------+-------------+------+-----+---------+----------------+
| tid   | int(4)      | NO   | PRI | NULL    | auto_increment |
| name  | varchar(30) | YES  |     | NULL    |                |
| subj1 | int(2)      | YES  |     | NULL    |                |
| subj2 | int(2)      | YES  |     | NULL    |                |
| subj3 | int(2)      | YES  |     | NULL    |                |
| total | int(3)      | YES  |     | NULL    |                |
| per   | int(3)      | YES  |     | NULL    |                |
+-------+-------------+------+-----+---------+----------------+
7 rows in set (0.00 sec)
```

SQL Trigger to problem statement.

```
create trigger stud_marks
before INSERT
on
Student
for each row
set Student.total = Student.subj1 + Student.subj2 + Student.subj3,
Student.per = Student.total * 60 / 100;
```

Above SQL statement will create a trigger in the student database in which whenever subjects marks are entered, before inserting this data into the database, trigger will compute those two values and insert with the entered values. i.e.,

```
mysql> insert into Student values(0, "ABCDE", 20, 20, 20, 0, 0);
Query OK, 1 row affected (0.09 sec)


mysql> select * from Student;
+-----+-------+-------+-------+-------+-------+------+
| tid | name  | subj1 | subj2 | subj3 | total | per  |
+-----+-------+-------+-------+-------+-------+------+
| 100 | ABCDE |    20 |    20 |    20 |    60 |   36 |
+-----+-------+-------+-------+-------+-------+------+
1 row in set (0.00 sec)
```