Formal Relational Query Languages

The Relational operations, The Tuple Relational Calculus, The Domain Relational Calculus.

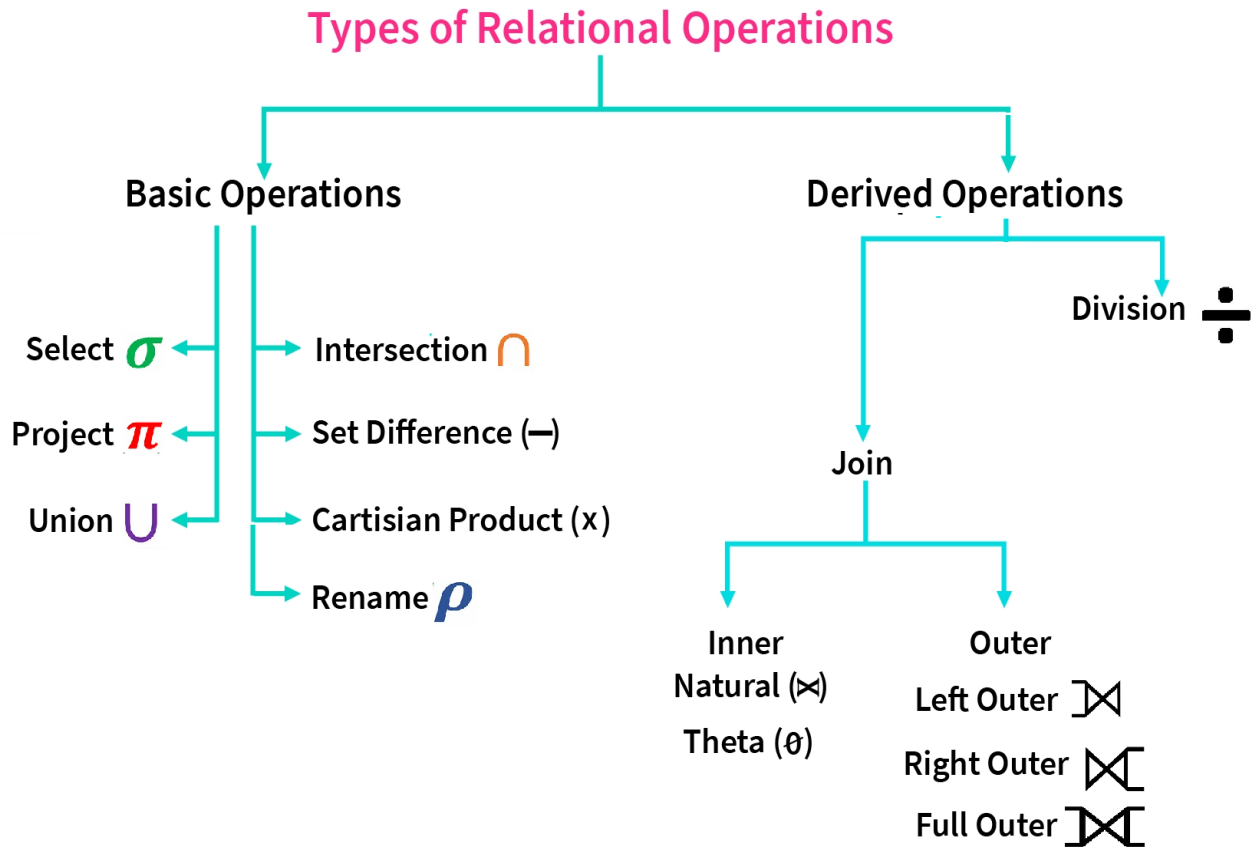
Relational Database Design

Features of Good Relational Designs, Atomic Domains and First Normal Form, Decomposition Using Functional Dependencies, Decomposition Using Multi valued Dependencies, BCNF.

The Relational Algebra

The Relational Algebra is a *procedural query language*. It consists of a set of operations that take one or two relations as input and produce a new relation as their result.

The Basic operations in the relational algebra are *select*, *project*, *union*, *set difference*, *Cartesian product rename*, and *set intersection*. In addition to the basic operations, there are several other operations—namely, N*atural Join* and *Division*



We present a number of sample queries using the following schema:

Sailors(sid: integer, sname: string, rating: integer, age: real)

Boats(bid: integer, bname: string, color: string)

Reserves(sid: integer, bid: integer, day: date)

The key fields are underlined, and the domain of each field is listed after the field name. Thus, **sid** is the key for Sailors, **bid** is the key for Boats, and all three fields together form the key for Reserves. Fields in an instance of one of these relations are referred to by name, or positionally, using the order in which they were just listed.

In several examples illustrating the relational algebra operators, we use the instances S1 and S2 (of Sailors) and R1 (of Reserves) shown in Figures 4.1, 4.2, and 4.3, respectively.

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

Figure 4.1 Instance $S1$ of Sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Figure 4.2 Instance $S2$ of Sailors

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

Figure 4.3 Instance $R1$ of Reserves

1. The Select Operation

Relational algebra includes operator to select rows from a relation **(σ).** This operation allow us to manipulate data in a single relation.

Consider the instance of the Sailors relation shown in Figure 4.2, denoted as S2. We can retrieve rows corresponding to expert sailors by using the σ operator.

The expression

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

Evaluates to the relation shown in Figure 4.4. The subscript **rating>8** specifies the selection criterion to be applied while retrieving tuples.

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

The selection operator **σ** specifies the tuples to retain through a selection condition.

In general, the selection condition is a Boolean combination (i.e., an expression using the logical connectives ∧ and ∨) of terms that have the form attribute op constant or attribute1 op attribute2, where op is one of the comparison operators <, <=, =, =, >=, or >.

2. The Project Operation

Relational algebra includes operator to project columns **(π).**

The projection operator **π** allows us to extract columns from a relation;

For example, we can find out all **sailor names** and **ratings** by using **π.**
The expression

π sname, rating (S2)

evaluates to the relation shown in Figure 4.5. The subscript sname, rating specifies the fields to be retained; the other fields are 'projected out.' The schema of the result of a projection is determined by the fields that are projected in the obvious way.

sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

For example,

We can compute the names and ratings of highly rated sailors by combining two of the Preceding queries.

The expression

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

Produces the result shown in Figure 4.7. It is obtained by applying the selection to S2 (to get the relation shown in Figure 4.4) and then applying the projection.

sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

3. The Union Operation

R ∪ S returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both). R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

Two relation instances are said to be union-compatible if the following conditions hold:

- they have the same number of the fields, and
- Corresponding fields, taken in order from left to right, have the same domains.

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

Figure 4.8 $S1 \cup S2$

4. The Intersection Operation

R ∩ S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
58	Rusty	10	35.0

Figure 4.9 $S1 \cap S2$

5. The Set-Difference Operation

R−S returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

Figure 4.10 $S1 - S2$

6. The Cartesian Product Operation

R × S returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S).
The result of **R × S** contains one tuple r, s (the concatenation of tuples r and s) for each pair of tuples **r ∈ R, s ∈ S**. The cross-product opertion is sometimes called **Cross product.**

(sid)	sname	rating	age	(sid)	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 $S1 \times R1$

7. **The Rename Operation**

The expression **ρ(R(F), E)** takes an arbitrary relational algebra expression E and returns an instance of a (new) relation called R. R contains the same tuples as the result of E and has the same schema as E, but some fields are renamed. The field names in relation R are the same as in E, except for fields renamed in the renaming list F , which is a list of terms having the form oldname → newname or position → newname.

For **ρ** to be well-defined, references to fields (in the form of oldnames or positions in the renaming list) may be unambiguous and no two fields in the result may have the same name. Sometimes we want to only rename fields or (re)name the relation; we therefore treat both R and F as optional in the use of **ρ.**

For example, the expression

ρ(C(1 → sid1, 5 → sid2), S1 × R1)

returns a relation that contains the tuples shown in Figure 4.11 and has the following schema: C(sid1: integer, sname: string, rating: integer, age: real, sid2: integer, bid: integer, day: dates).

(sid)1	sname	rating	age	(sid)2	bid	day
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

Figure 4.11 $S1 \times R1$

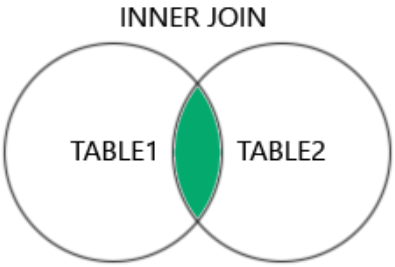
Derived Operations

1. Natural Join
2. Theta Join
3. Left Outer Join
4. Right Outer Join
5. Full Outer Join
6. Division

1. Natural Join Operation

The *natural join* is a binary operation that allows us to combine certain selections and a Cartesian product into one operation. It is denoted by the **join** symbol ⋈. The natural-join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas, and finally removes duplicate attributes.

Preferably Natural Join is performed on the foreign key.
Notation : **R ⋈ S**
Where R is the first relation
S is the second relation



Example ***"Find the names of all instructors together with the course id of all courses they taught."***. We express this query by using the natural join as follows:

$$\Pi_{name,\ course\_id}\ (instructor \bowtie teaches)$$

2. Theta Join Operation

The *theta join* operation is a variant of the natural-join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations r (R) and s(S), and let **"theta"(θ)** be a predicate on attributes in the schema $R \cup S$.

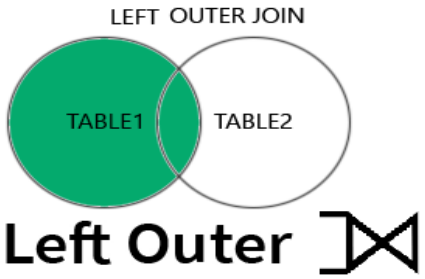
The **theta join** operation $r \bowtie_\theta s$ is defined as follows:

Notation : $R \bowtie_\theta S = \sigma_\theta(R \times S)$

Where R is the first relation, S is the second relation

3. Left Outer Join Operation

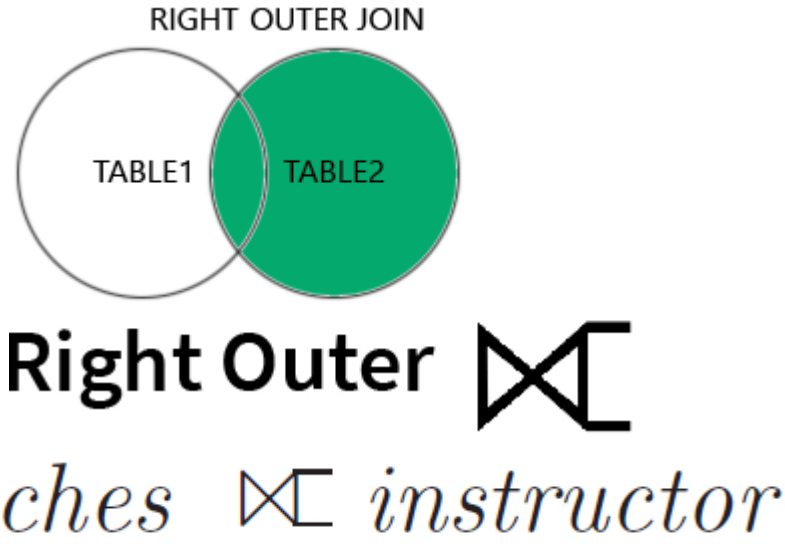
The **left outer join** (⟕) takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with **null values** for all other attributes from the right relation, and adds them to the result of the natural join.



$$instructor \ ⟕ \ teaches$$

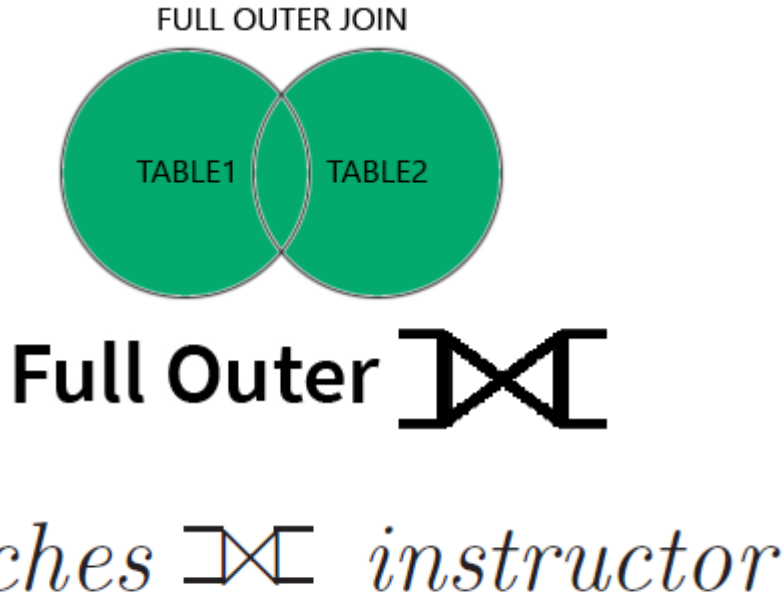
4. Right Outer Join Operation

The **right outer join** (⟖) is symmetric with the left outer join: It pads tuples from the right relation that did not match any from the left relation with **nulls** and adds them to the result of the natural join.



5. Full Outer Join Operation

The **full outer join**(⟗) does both the left and right outer join operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.



6. Division Operation

Consider two relation instances **A** and **B** in which A has (exactly) two fields **x and y** and B has just one field **y**, with the same domain as in A. We define the division operation **A/B** as the set of all x values (in the form of unary tuples) such that for every y value in (a tuple of) B, there is a **tuple (x,y) in A**.

Division is illustrated in Figure 4.14. It helps to think of **A as a relation listing the parts supplied by suppliers** and of the **B relations as listing parts**.

A/Bi computes suppliers who supply all parts listed in relation instance Bi.

A

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

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Figure 4.14 Examples Illustrating Division

Examples of Algebra Queries

We use the Sailors, Reserves, and Boats schema

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

Figure 4.15 An Instance $S3$ of Sailors

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

Figure 4.16 An Instance $R2$ of Reserves

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

Figure 4.17 An Instance $B1$ of Boats

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

   This query can be written as follows:

$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

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

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

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

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

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

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

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

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

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

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'} Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cup Tempgreen) \bowtie Sailors)$$

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

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$
$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$
$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$
$$\pi_{sname1}\sigma_{(sid1=sid2)\wedge(bid1\neq bid2)} Reservationpairs$$

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

$$\pi_{sid}(\sigma_{age>20} Sailors) -$$
$$\pi_{sid}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

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

$$\rho(Tempsids, (\pi_{sid,bid} Reserves)/(\pi_{bid} Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

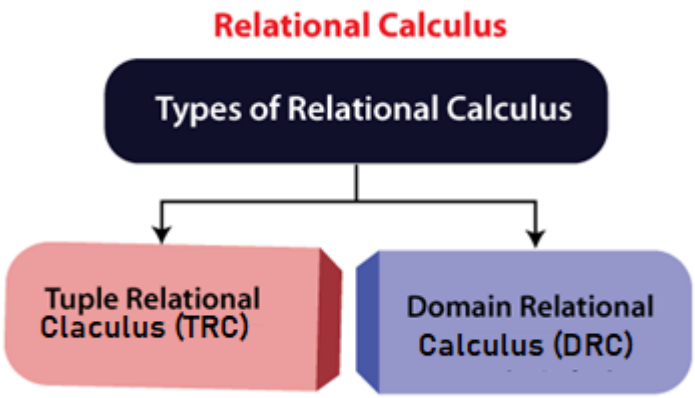
10. Find the names of sailors who have reserved all boats called Interlake.

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

RELATIONAL CALCULUS

Relational calculus is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative, in that it allows us to describe the set of answers without being explicit about how they should be computed.

Relational calculus has had a big influence on the design of commercial query languages such as SQL and, especially, Query By- Example (QBE).



The variant of the calculus we present in detail is called the **Tuple Relational Calculus (TRC).** Variables in TRC take on tuples as values.

In another variant, called the **Domain Relational Calculus (DRC)**, the variables range over field values.

- **TRC** has had more of an influence on **SQL**,
- **DRC** has strongly influenced **QBE**.

- Relational Calculus in DBMS tells us what we want from the database and not how to get that.
- Relational Calculus is a Declarative Language.
- TRC uses tuple variable and checks every Row with the Predicate expression condition.
- DRC uses domain variables and returns the required attribute or column based on the condition.

The Tuple Relational Calculus

Tuple Relational Calculus (TRC) is a non-procedural query language used in relational database management systems (RDBMS) to retrieve data from tables. TRC is based on the concept of tuples, which are ordered sets of attribute values that represent a single row or record in a database table.

TRC is a declarative language, meaning that it specifies what data is required from the **database**, rather than how to retrieve it. TRC queries are expressed as logical formulas that describe the desired tuples.

Syntax: The basic syntax of TRC is as follows:

{ T | P(T) }

where T is a **tuple variable** and P(T) is a **logical formula** that describes the conditions that the tuples in the result must satisfy. The **curly braces {}** are used to indicate that the expression is a set of tuples.

Syntax of TRC Queries

We now define these concepts formally, beginning with the notion of a formula.
Let Rel be a relation name, R and S be tuple variables, a be an attribute of R, and b be an attribute of S. Let op denote an operator in the set {<, >, = , ≤, ≥, =}.
An atomic formula is one of the following:

- **R ∈ Rel**
- **R.a op S.b**
- **R.a op constant, or constant op R.a**

A formula is recursively defined to be one of the following, where p and q are themselves formulas and p(R) denotes a formula in which the variable R appears:

- any atomic formula
- **¬p, p ∧ q, p ∨ q, or p ⇒ q**
- **∃R(p(R)),** where R is a tuple variable
- **∀R(p(R)),** where R is a tuple variable

Examples of TRC Queries

1. **Find all sailors with a rating above 7.**

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

2. **Find the names and ages of sailors with a rating above 7.**

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

3. **Find the sailor name, boat id, and reservation date for each reservation.**

$$\{P \mid \exists R \in Reserves \ \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

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

$$\{P \mid \exists S \in Sailors \ \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

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

$$\{P \mid \exists S \in Sailors \ \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

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

$$\{P \mid \exists S \in Sailors \ \exists R \in Reserves (S.sid = R.sid \wedge P.sname = S.sname)\}$$

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

$$\{P \mid \exists S \in Sailors \ \exists R1 \in Reserves \ \exists R2 \in Reserves (S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

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

$$\{P \mid \exists S \in Sailors \ \forall B \in Boats (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

9. **Find sailors who have reserved all red boats.**

$$\{S \mid S \in Sailors \wedge \forall B \in Boats (B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

Domain Relational Calculus

Domain Relational Calculus is a non-procedural query language equivalent in power to Tuple Relational Calculus. Domain Relational Calculus provides only the description of the query but it does not provide the methods to solve it. In Domain Relational Calculus, a query is expressed as,

$$\{ < x_1, x_2, x_3, ..., x_n > | P (x_1, x_2, x_3, ..., x_n) \}$$

where, $< x_1, x_2, x_3, ..., x_n >$ represents resulting domains variables and $P (x_1, x_2, x_3, ..., x_n)$ represents the condition or formula equivalent to the Predicate calculus.

A DRC formula is defined in a manner very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables.

Let **op** denote an operator in the set {<, >, =, ≤, ≥, =} and let X and Y be domain variables. An atomic formula in DRC is one of the following:

- **x1, x2, . . . , xn ∈ Rel**, where Rel is a relation with n attributes; each
- **xi, 1 ≤ i ≤ n** is either a variable or a constant
- **X op Y**
- **X op constant, or constant op X**

A formula is recursively defined to be one of the following, where p and q are themselves formulas and p(X) denotes a formula in which the variable X appears:

- any atomic formula
- **¬p, p ∧ q, p ∨ q, or p ⇒ q**
- **∃X(p(X)),** where X is a domain variable
- **∀ X(p(X)),** where X is a domain variable

Examples of DRC Queries

1. Find all sailors with a rating above 7.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

The condition **I, N, T , A ∈ Sailors** ensures that the domain variables I, N , T , and A are restricted to be fields of the same tuple. In comparison with the TRC query, we can say **T > 7** instead of **S.rating > 7**, but we must specify the tuple I, N, T , A in the result, rather than just S.

2. Find the names of sailors who have reserved boat 103.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Ir, Br, D(\langle Ir, Br, D\rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

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

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists\langle I, Br, D\rangle \in Reserves \wedge \exists\langle Br, BN, 'red'\rangle \in Boats)\}$$

4. Find the names of sailors who have reserved at least two boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \exists Br1, Br2, D1, D2(\langle I, Br1, D1\rangle \in Reserves \wedge \langle I, Br2, D2\rangle \in Reserves \wedge Br1 \neq Br2))\}$$

5. Find the names of sailors who have reserved all boats.

$$\{\langle N\rangle \mid \exists I, T, A(\langle I, N, T, A\rangle \in Sailors \wedge \forall B, BN, C(\neg(\langle B, BN, C\rangle \in Boats) \vee (\exists\langle Ir, Br, D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

6. Find sailors who have reserved all red boats.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge \forall \langle B, BN, C\rangle \in Boats$$
$$(C = 'red' \Rightarrow \exists \langle Ir, Br, D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

Relational Database Design

Features of Good Relational Designs, Atomic Domains and First Normal Form, Decomposition Using Functional Dependencies, Decomposition Using Multi valued Dependencies, BCNF.

Features of Good Relational Designs

- Relational design is essential for efficient data retrieval and manipulation in a database.
- Maintaining a good relational design is crucial for the organization and maintainability of a database.
- Constraints, such as uniqueness and dependencies, should be carefully defined to ensure data integrity.
- Reducing the number of schemas in a database design can improve performance and simplify management.
- Decomposing larger schemas into smaller ones helps eliminate redundant information and loss of data.
- Lossless decomposition ensures that no data is lost during the decomposition process.
- Functional dependencies play a key role in preserving relationships and constraints in a relational schema.

Different Types of Database Keys

1. Unique Key
2. Primary Key
3. Foreign Key
4. Super Key
5. Candidate Key
6. Alternate Key or Secondary Key
7. Composite Key or Compound Key

1. Unique Key

A unique key in DBMS is used to uniquely identify a tuple in a table and is used to prevent duplicity of the values in a table. A unique key can have one **NULL** value as its value.

2. Primary Key

- It is a unique key.
- It can identify only one tuple (a record) at a time.
- It has no duplicate values, it has unique values.
- It cannot be NULL.
- Primary keys are not necessarily to be a single column; more than one column can also be a primary key for a table.

3. Foreign Key

- It is a key it acts as a primary key in one table and it acts as secondary key in another table.
- It combines two or more relations (tables) at a time.
- They act as a cross-reference between the tables.

4. Super Key

We can define a super key as a set of those keys that identify a row or a tuple uniquely. The word super denotes the superiority of a key. Thus, a super key is the superset of a key known as a **Candidate Key**. It means a **candidate key** is obtained from a super key only.

The **EMPLOYEE_DETAIL** table is given below that will help you understand better:

Emp_SSN	Emp_Id	Emp_name	Emp_email
11051	01	John	john@email.com
19801	02	Merry	merry@email.com
19801	03	Riddle	riddle@email.com
41201	04	Cary	cary@email.com

So, from the above table, we conclude the following set of the super keys:

Set of super keys obtained

{ Emp_SSN }
{ Emp_Id }
{ Emp_email }
{ Emp_SSN, Emp_Id }
{ Emp_Id, Emp_name }
{ Emp_SSN, Emp_Id, Emp_email }
{ Emp_SSN, Emp_name, Emp_Id }

5. Candidate Key

A candidate key is a subset of a super key set where the key which contains no redundant attribute is none other than a **Candidate Key**. In order to select the candidate keys from the set of super key, we need to look at the super key set.

Candidate Keys :

Emp_SSN
Emp_Id
Emp_email

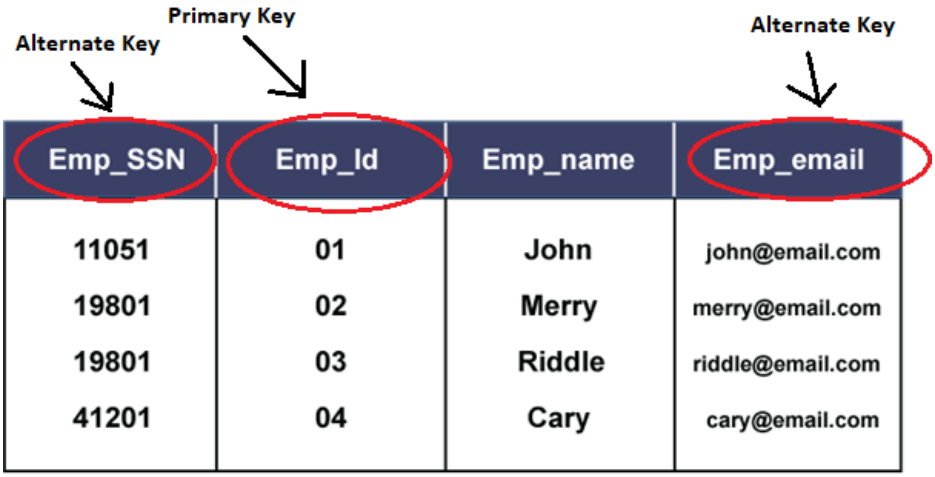
6. Alternate Key or Secondary Key

An alternate key is the secondary candidate key that contains all the property of a candidate key but is an alternate option.
The EMPLOYEE_DETAIL table is given below that will help you understand better:

Emp_SSN	Emp_Id	Emp_name	Emp_email
11051	01	John	john@email.com
19801	02	Merry	merry@email.com
19801	03	Riddle	riddle@email.com
41201	04	Cary	cary@email.com

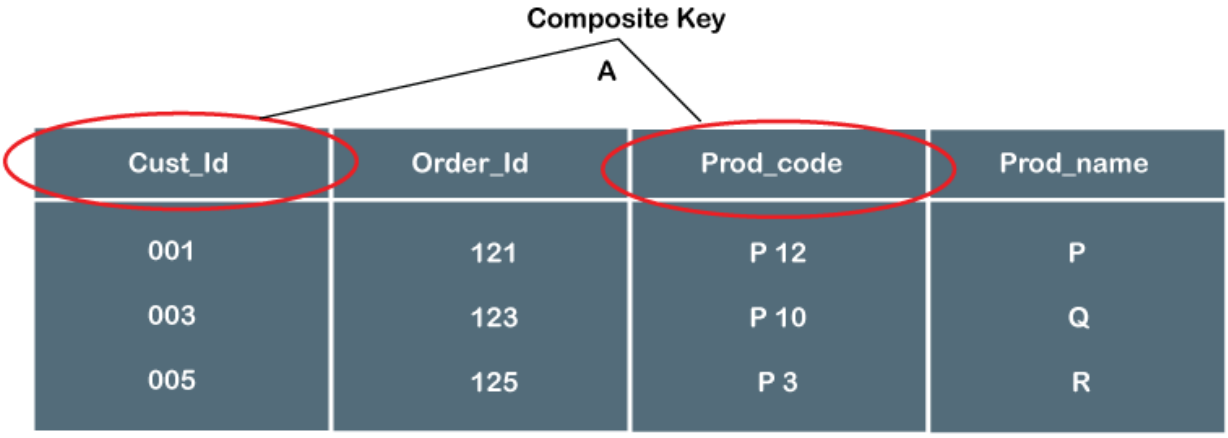
So from the above table, there are the following Candidate keys:

These are the candidate keys we concluded from the above attributes. Now, we have to choose one primary key, which is the most appropriate out of the three, and it is **Emp_Id**. *So, the primary key is Emp_Id.* Now, the remaining two candidate keys are **Emp_SSN** and **Emp_email**. Therefore, Emp_SSN and Emp_Email are the alternate keys.



7. Composite Key or Compound Key

Two or more attributes together form a composite key that can uniquely identify a tuple in a table. Such a key is also known as **Compound Key.**



From the above table, we found that no attribute is available that alone can identify a record in the table and can become a **Primary Key**. However, the combination of some attributes can form a key and can identify a tuple in the table. In the above example, **Cust_Id** and **Prod_code** can together form a primary key because they alone are not able to identify a tuple, but together they can do so.

Decomposition in DBMS

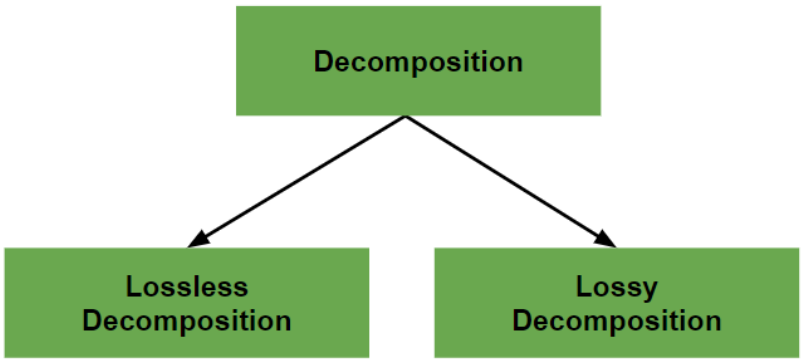
Decomposition refers to the division of tables into multiple tables to produce consistency in the data.

We perform decomposition in DBMS when we want to process a particular data set. It is performed in a database management system when we need to ensure consistency and remove anomalies and duplicate data present in the database. When we perform decomposition in DBMS, we must try to ensure that no information or data is lost.

Types of Decomposition

There are two types of Decomposition:

1. Lossless Decomposition
2. Lossy Decomposition



1. Lossless Decomposition

The process in which where we can regain the original relation R with the help of joins from the multiple relations formed after decomposition. This process is termed as lossless decomposition. It is used to remove the redundant data from the database while retaining the useful information. The lossless decomposition tries to ensure following things:

- While regaining the original relation, no information should be lost.
- If we perform join operation on the sub-divided relations, we must get the original relation.

Example:

There is a relation called R(A, B, C)

A	B	C
55	16	27
48	52	89

Now we decompose this relation into two sub relations R1 and R2

R1(A, B)

A	B
55	16
48	52

R2(B, C)

B	C
16	27
52	89

After performing the Join operation we get the same original relation

A	B	C
55	16	27
48	52	89

2. Lossy Decomposition

As the name suggests, lossy decomposition means when we perform join operation on the sub-relations it doesn't result to the same relation which was decomposed. After the join operation, we always found some extraneous tuples. These extra tuples genrates difficulty for the user to identify the original tuples.

Example:

We have a relation **R(A, B, C)**

A	B	C
1	2	1
2	5	3
3	3	3

Now , we decompose it into sub-relations R1 and R2

R1(A, B)

A	B
1	2
2	5
3	3

R2(B, C)

B	C
2	1
5	3
3	3

Now After performing join operation

A	B	C
1	2	1
2	5	3
2	3	3
3	5	3
3	3	3

Properties of Decomposition

- **Lossless:** All the decomposition that we perform in Database management system should be lossless. All the information should not be lost while performing the join on the sub-relation to get back the original relation. It helps to remove the redundant data from the database.
- **Dependency Preservation:** Dependency Preservation is an important technique in database management system. It ensures that the functional dependencies between the entities is maintained while performing decomposition. It helps to improve the database efficiency, maintain consistency and integrity.
- **Lack of Data Redundancy:** Data Redundancy is generally termed as duplicate data or repeated data. This property states that the decomposition performed should not suffer redundant data. It will help us to get rid of unwanted data and focus only on the useful data or information.

Functional Dependency

In relational database management, functional dependency is a concept that specifies the relationship between two sets of attributes where one attribute determines the value of another attribute. It is denoted as **X → Y**, where the attribute set on the left side of the arrow, **X** is called **Determinant**, and **Y** is called the **Dependent**.

A functional dependency occurs when one attribute uniquely determines another attribute within a relation. It is a constraint that describes how attributes in a table relate to each other. If attribute A functionally determines attribute B we write this as the **A→B**.

Functional dependencies are used to mathematically express relations among database entities.

Example:

roll_no	name	dept_name	dept_building
42	abc	CO	A4
43	pqr	IT	A3
44	xyz	CO	A4
45	xyz	IT	A3
46	mno	EC	B2
47	jkl	ME	B2

From the above table we can conclude some valid functional dependencies:

- roll_no → { name, dept_name, dept_building },→ Here, roll_no can determine values of fields name, dept_name and dept_building, hence a valid Functional dependency
- roll_no → dept_name , Since, roll_no can determine whole set of {name, dept_name, dept_building}, it can determine its subset dept_name also.

- dept_name → dept_building , Dept_name can identify the dept_building accurately, since departments with different dept_name will also have a different dept_building
- More valid functional dependencies: roll_no → name, {roll_no, name} ⇢ {dept_name, dept_building}, etc.

Here are some invalid functional dependencies:

- name → dept_name Students with the same name can have different dept_name, hence this is not a valid functional dependency.
- dept_building → dept_name There can be multiple departments in the same building. Example, in the above table departments ME and EC are in the same building B2, hence dept_building → dept_name is an invalid functional dependency.
- More invalid functional dependencies: name → roll_no, {name, dept_name} → roll_no, dept_building → roll_no, etc.

Armstrong's axioms/properties of functional dependencies:

1. **Reflexivity:** If Y is a subset of X, then X→Y holds by reflexivity rule
   Example, {roll_no, name} → name is valid.
2. **Augmentation:** If X → Y is a valid dependency, then XZ → YZ is also valid by the augmentation rule.
   Example, {roll_no, name} → dept_building is valid, hence {roll_no, name, dept_name} → {dept_building, dept_name} is also valid.
3. **Transitivity**: If X → Y and Y → Z are both valid dependencies, then X→Z is also valid by the Transitivity rule.
   Example, roll_no → dept_name & dept_name → dept_building, then roll_no → dept_building is also valid.

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

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.

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 !**)

4. 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.

5. **Fully Functional Dependency**

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes. If a relation R has attributes X, Y, Z with the dependencies X->Y and X->Z which states that those dependencies are fully functional.

6. **Partial Functional Dependency**

In partial functional dependency a non key attribute depends on a part of the composite key, rather than the whole key. If a relation R has attributes X, Y, Z where X and Y are the composite key and Z is non key attribute. Then X->Z is a partial functional dependency in RBDMS.

Advantages of Functional Dependencies

Functional dependencies having numerous applications in the field of database management system. Here are some applications listed below:

1. **Data Normalization**

Data normalization is the process of organizing data in a database in order to minimize redundancy and increase data integrity. Functional dependencies play an important part in data normalization. With the help of functional dependencies we are able to identify the primary key, candidate key in a table which in turns helps in normalization.

2. **Query Optimization**

With the help of functional dependencies we are able to decide the connectivity between the tables and the necessary attributes need to be projected to retrieve the required data from the tables. This helps in query optimization and improves performance.

3. **Consistency of Data**

Functional dependencies ensures the consistency of the data by removing any redundancies or inconsistencies that may exist in the data. Functional dependency ensures that the changes made in one attribute does not affect inconsistency in another set of attributes thus it maintains the consistency of the data in database.

4. **Data Quality Improvement**
   Functional dependencies ensure that the data in the database to be accurate, complete and updated. This helps to improve the overall quality of the data, as well as it eliminates errors and inaccuracies that might occur during data analysis and decision making, thus functional dependency helps in improving the quality of data in database.

Normal Forms in DBMS

Normalization is the process of minimizing **redundancy** from a relation or set of relations. Redundancy in relation may cause insertion, deletion, and updation anomalies. So, it helps to minimize the redundancy in relations. **Normal forms** are used to eliminate or reduce redundancy in database tables.

- 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.

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.

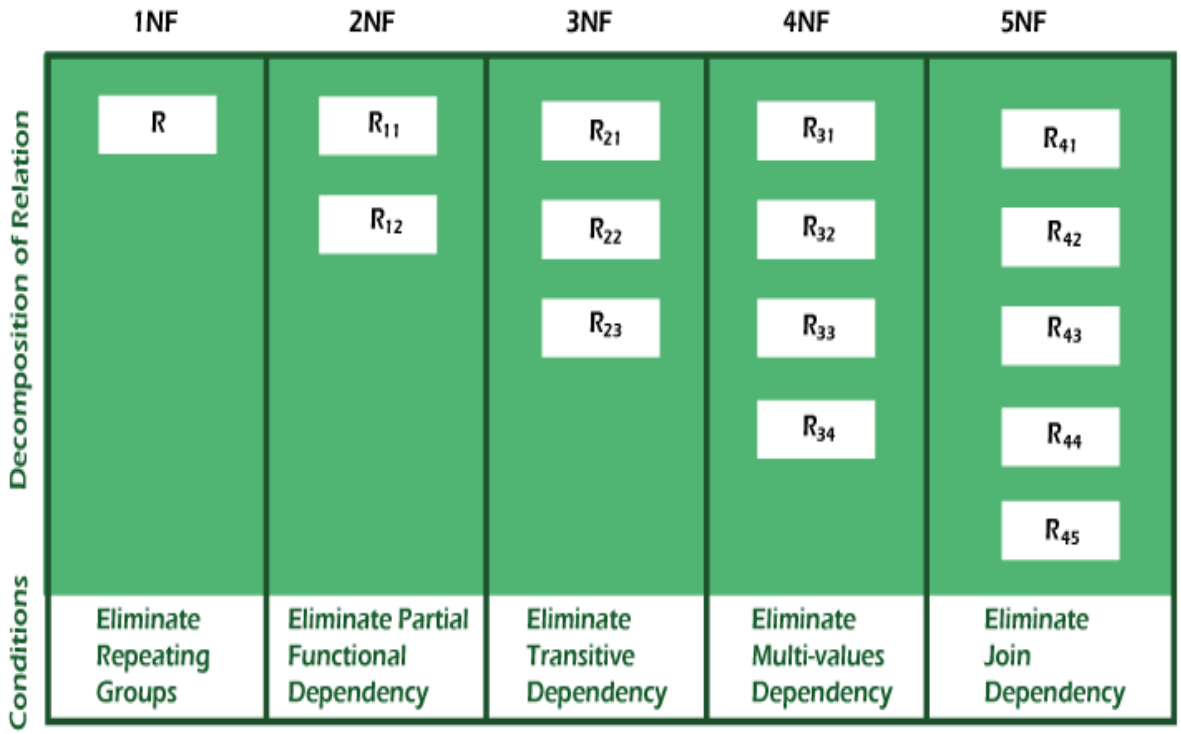
Data modification anomalies can be categorized into three types:

1. **Insertion Anomaly:** Insertion Anomaly refers to when one cannot insert a new tuple into a relationship due to lack of data.
2. **Deletion Anomaly:** The delete anomaly refers to the situation where the deletion of data results in the unintended loss of some other important data.
3. **Updation 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.

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 transitive 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.

1. 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

2. 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

3. 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 on

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

4. Boyce Codd Normal Form (BCNF)

- BCNF is the advance version stricter than 3NF.
- A table is in BCNF if every functional dependency X → Y, X is the super key of the table. of 3NF. It is
- 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
364	UK

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.

What is the Difference Between 3NF and BCNF?

Parameter	3NF	BCNF
Definition	A relation is in 3NF if it is in 2NF, and every non-prime attribute is fully functionally dependent on every candidate key.	A relation is in BCNF if every determinant is a candidate key.
Dependency	Eliminates transitive dependencies.	Eliminates all types of dependencies except trivial functional dependencies.
Candidate Keys	Allows for multiple candidate keys.	Allows only one candidate key.
Redundancy	It may contain some redundant data.	Does not contain any redundant data.
Normal Form Level	Stricter than 2NF but less strict than BCNF.	Stricter than 3NF.
Anomalies	Prevents insertion, update, and deletion anomalies.	Prevents all types of anomalies, including those prevented by 3NF.
Decomposition	It may require further decomposition.	No further decomposition is required.
Complexity	• Less complex than BCNF.	More complex than 3NF.
Preservation	Preserves data integrity partially.	Preserves data integrity completely.
Practical Use	Widely used in database design.	Used in specific cases where data integrity is critical.

Multivalued Dependency

- Multivalued dependency occurs when two attributes in a table are independent of each other but, both depend on a third attribute.
- A multivalued dependency consists of at least two attributes that are dependent on a third attribute that's why it always requires at least three attributes.

Example: Suppose there is a bike manufacturer company which produces two colors(white and black) of each model every year.

BIKE_MODEL	MANUF_YEAR	COLOR
M2011	2008	White
M2001	2008	Black
M3001	2013	White
M3001	2013	Black
M4006	2017	White
M4006	2017	Black

Here columns COLOR and MANUF_YEAR are dependent on BIKE_MODEL and independent of each other.

In this case, these two columns can be called as multivalued dependent on BIKE_MODEL. The representation of these dependencies is shown below:

1. BIKE_MODEL → → MANUF_YEAR
2. BIKE_MODEL → → COLOR

This can be read as "BIKE_MODEL multidetermined MANUF_YEAR" and "BIKE_MODEL multidetermined COLOR".

5. 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

6. 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

Advantages of Normal Form

- **Reduced data redundancy:** Normalization helps to eliminate duplicate data in tables, reducing the amount of storage space needed and improving database efficiency.
- **Improved data consistency:** Normalization ensures that data is stored in a consistent and organized manner, reducing the risk of data inconsistencies and errors.
- **Simplified database design:** Normalization provides guidelines for organizing tables and data relationships, making it easier to design and maintain a database.
- **Improved query performance:** Normalized tables are typically easier to search and retrieve data from, resulting in faster query performance.
- **Easier database maintenance:** Normalization reduces the complexity of a database by breaking it down into smaller, more manageable tables, making it easier to add, modify, and delete data.

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.