Codd's 12 Rules

Rule 1: Information Rule

The data stored in a database, may it be user data or metadata, must be a value of some table cell. Everything in a database must be stored in a table format.

Rule 2: Guaranteed Access Rule

Every single data element (value) is guaranteed to be accessible logically with a combination of table-name, primary-key (row value), and attribute-name (column value). No other means, such as pointers, can be used to access data.

Rule 3: Systematic Treatment of NULL Values

The NULL values in a database must be given a systematic and uniform treatment. This is a very important rule because a NULL can be interpreted as one the following − data is missing, data is not known, or data is not applicable.

Rule 4: Active Online Catalog

The structure description of the entire database must be stored in an online catalog, known as **data dictionary**, which can be accessed by authorized users. Users can use the same query language to access the catalog which they use to access the database itself.

Rule 5: Comprehensive Data Sub-Language Rule

A database can only be accessed using a language having linear syntax that supports data definition, data manipulation, and transaction management operations. This language can be used directly or by means of some application. If the database allows access to data without any help of this language, then it is considered as a violation. Example SQL

Rule 6: View Updating Rule

All the views of a database, which can theoretically be updated, must also be updatable by the system.

Rule 7: High-Level Insert, Update, and Delete Rule

A database must support high-level insertion, updation, and deletion. This must not be limited to a single row, that is, it must also support union, intersection and minus operations to yield sets of data records.

Rule 8: Physical Data Independence

The data stored in a database must be independent of the applications that access the database. Any change in the physical structure of a database must not have any impact on how the data is being accessed by external applications.

Rule 9: Logical Data Independence

The logical data in a database must be independent of its user's view (application). Any change in logical data must not affect the applications using it. For example, if two tables are merged or one is split into two different tables, there should be no impact or change on the user application. This is one of the most difficult rule to apply.

Rule 10: Integrity Independence

A database must be independent of the application that uses it. All its integrity constraints can be independently modified without the need of any change in the application. This rule makes a database independent of the front-end application and its interface.

Rule 11: Distribution Independence

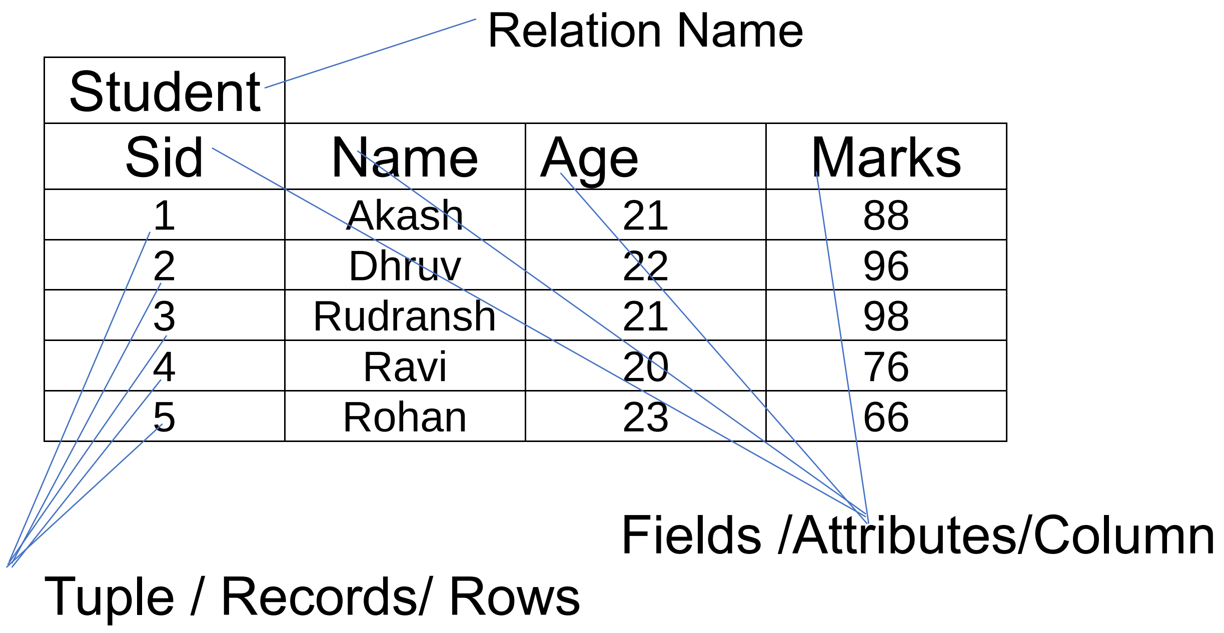
The end-user must not be able to see that the data is distributed over various locations. Users should always get the impression that the data is located at one site only. This rule has been regarded as the foundation of distributed database systems.

Rule 12: Non-Subversion Rule

If a system has an interface that provides access to low-level records, then the interface must not be able to subvert the system and bypass security and integrity constraints.

Relation Data Model

Concepts



Student			
Sid	Name	Age	Marks
1	Akash	21	88
2	Dhruv	22	96
3	Rudransh	21	98
4	Ravi	20	76
5	Rohan	23	66

Create table student (sid number(2), Name varchar2(10), Age Number(2), Marks number(2));

Tables − In relational data model, relations are saved in the format of Tables. This format stores the relation among entities. A table has rows and columns, where rows represents records and columns represent the attributes.

Relation instance – refers to actual table

Relation schema − A relation schema describes the relation name (table name), attributes, and their names.

Domain of Attributes – set of permitted values that can be assigned to the attribute.

Degree of Relational Schema – is the number of attributes it contains

Cardinality of Relational Schema –is the number of tuples it contains

Keys in DBMS

Student

ID	Roll_No	Name	Enroll_No	Deptid
1	603	A	AX1	1
2	604	B	AX2	1
3	605	A	AX3	1
4	606	D		3

Department

Deptid	D_Name	Location
1	CSE	GF
2	IT	FF
3	ECE	SF

Note : every candidate key is a super key, but every super key may or may not be candidate key.

Super Key – A super key is a set of one of more columns (attributes) to **uniquely** identify rows in a table.

Candidate Key – A super key with no redundant attribute(minimal set) is known as candidate key

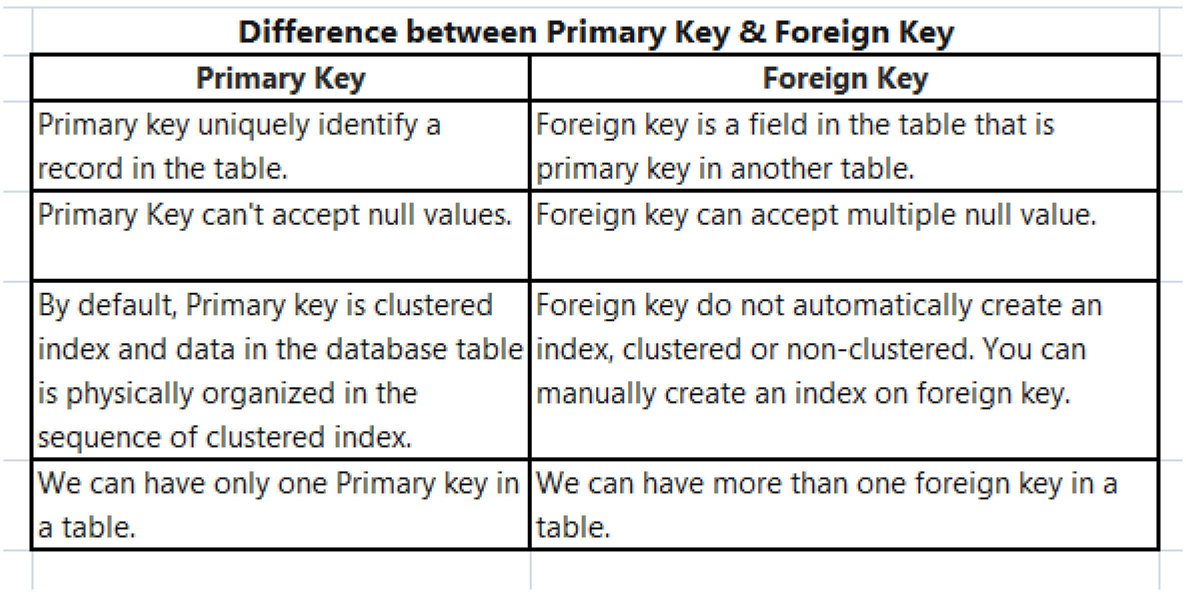
Primary Key – A primary is a column or set of columns in a table that uniquely identifies tuples (rows) in that table.

Alternate Key – Out of all candidate keys, only one gets selected as primary key, remaining keys are known as alternate or secondary keys.

Foreign Key – Foreign keys are the columns of a table that points to the primary key of another table. They act as a cross-reference between tables.

Unique Key – A primary is a column or set of columns in a table that uniquely values in tuples (rows) of that table, it can have one null.

Composite Key – A key that consists of more than one attribute to uniquely identify rows (also known as records & tuples) in a table is called composite key.



Difference between Primary Key & Foreign Key

Primary Key	Foreign Key
Primary key uniquely identify a record in the table.	Foreign key is a field in the table that is primary key in another table.
Primary Key can't accept null values.	Foreign key can accept multiple null value.
By default, Primary key is clustered index and data in the database table is physically organized in the sequence of clustered index.	Foreign key do not automatically create an index, clustered or non-clustered. You can manually create an index on foreign key.
We can have only one Primary key in a table.	We can have more than one foreign key in a table.

Integrity Constraints in DBMS

- Constraints or nothing but the rules that are to be followed while entering data into columns of the database table
- Constraints ensure that data entered by the user into columns must be within the criteria specified by the condition
- For example, if you want to maintain only unique IDs in the employee table or if you want to enter only age under 18 in the student table etc
- We have 5 types of key constraints
    - `NOT NULL:` ensures that the specified ***column doesn't contain a NULL value.***
    - `UNIQUE :` ***provides a unique/distinct values*** to specified columns.
    - `DEFAULT:` ***provides a default value to a column*** if none is specified.
    - `CHECK :` ***checks for the predefined conditions before inserting*** the data inside the table.
    - `PRIMARY KEY:` it ***uniquely identifies a row*** in a table.
    - `FOREIGN KEY:` ensures ***referential integrity*** of the relationship
    - `Domain/ENTITY Constraint`

create table student (ID NUMBER(3) primary key, ROLL_NO NUMBER(5), NAME VARCHAR2(10) not null, ENROLL_NO VARCHAR CHAR(3), DEPTID NUMBER(1) , **centre varchar2(5) check(centre in ('del', 'noi', 'ggr'),marks number(2) default 0, age number(2) check age<25**);

SELECT NAME, ID FROM STUDENT

Student

ID	Name	Age	Marks	Centre	Centre	Centre
1	x	23		Del	noi	
2	v	19		noi		
5	N	20		del		
3	b	18		ggr		

4	N	20		del		

Id	center

Properties of relational databases

- **Values are atomic.**
- **All of the values in a column have the same data type. → DOMAIN CONSTRAINT**
- **Each row is unique. → PRIMARY CONTRAINT → INDICES**
- **The sequence of columns is insignificant.**
- **The sequence of rows is insignificant.**
- **Each column has a unique name WITHIN ONE TABLE.**
- **Integrity constraints maintain data consistency across multiple tables.**

Relational Algebra in DBMS

Relational algebra is a **procedural** query language that works on relational model. The purpose of a query language is to retrieve data from database or perform various operations such as insert, update, delete on the data. When I say that relational algebra is a procedural query language, it means that it tells what data to be retrieved and how to be retrieved.

On the other hand relational calculus is a non-procedural query language, which means it tells what data to be retrieved but doesn't tell how to retrieve it.

Types of operations in relational algebra

We have divided these operations in two categories:
1. Basic Operations
2. Derived Operations

Basic/Fundamental Operations:

1. Select (σ)
2. Project (∏)
3. Union (∪)
4. Set Difference (-)
5. Cartesian product (X)
6. Rename (ρ)

Derived Operations:

1. Natural Join (⋈)
2. Left, Right, Full outer join (⟕, ⟖, ⟗)
3. Intersection (∩)
4. Division (÷)

Select Operator (σ)

Select Operator is denoted by sigma (σ) and it is used to find the tuples (or rows) in a relation (or table) which satisfy the given condition.

If you understand little bit of SQL then you can think of it as a where clause in SQL, which is used for the same purpose.

Syntax of Select Operator (σ)

```
σ Condition/Predicate(Relation/Table name)
```

Select Operator (σ) Example

```
Table: CUSTOMER
---------------

Customer_Id     Customer_Name      Customer_City
-----------     -------------      -------------
C10100           Steve              Agra
C10111           Raghu              Agra
C10115           Chaitanya          Noida
```

```
C10117          Ajeet              Delhi
C10118          Carl               Delhi
```

Query:

```
σ Customer_City="Agra" (CUSTOMER)
```

Output:

```
Customer_Id   Customer_Name    Customer_City
-----------   -------------    -------------
C10100        Steve            Agra
C10111        Raghu            Agra
```

Project Operator (∏)

Project operator is denoted by ∏ symbol and it is used to select desired columns (or attributes) from a table (or relation).

Project operator in relational algebra is similar to the Select statement in SQL.

Syntax of Project Operator (∏)

```
∏ column_name1, column_name2, ...., column_nameN(table_name)
```

Project Operator (∏) Example

In this example, we have a table CUSTOMER with three columns, we want to fetch only two columns of the table, which we can do with the help of Project Operator ∏.

```
Table: CUSTOMER

Customer_Id     Customer_Name      Customer_City
-----------     -------------      -------------
C10100           Steve              Agra
C10111           Raghu              Agra
C10115           Chaitanya          Noida
C10117           Ajeet              Delhi
C10118           Carl               Delhi
```

Query:

```
∏ Customer_Name, Customer_City (CUSTOMER)
```

Output:

```
Customer_Name      Customer_City
-------------      -------------
Steve              Agra
Raghu              Agra
Chaitanya          Noida
Ajeet              Delhi
Carl               Delhi
```

Rename (ρ)

Rename (ρ) operation can be used to rename a relation or an attribute of a relation.
Rename (ρ) Syntax:
ρ(new_relation_name, old_relation_name)

Rename (ρ) Example

Lets say we have a table customer, we are fetching customer names and we are renaming the resulted relation to CUST_NAMES.

Table: CUSTOMER

```
Customer_Id     Customer_Name      Customer_City
-----------     -------------      -------------
C10100           Steve              Agra
C10111           Raghu              Agra
C10115           Chaitanya          Noida
C10117           Ajeet              Delhi
C10118           Carl               Delhi
```

Query:

```
ρ(CUST_NAMES, ∏(Customer_Name)(CUSTOMER))
```

Output:

```
CUST_NAMES
----------
Steve
Raghu
Chaitanya
Ajeet
Carl
```

Union Operator (∪)

Union operator is denoted by ∪ symbol and it is used to select all the rows (tuples) from two tables (relations).

Lets discuss union operator a bit more. Lets say we have two relations R1 and R2 both have same columns and we want to select all the tuples(rows) from these relations then we can apply the union operator on these relations.

Note: The rows (tuples) that are present in both the tables will only appear once in the union set. In short you can say that there are no duplicates present after the union operation.

Syntax of Union Operator (∪)

```
table_name1 ∪ table_name2
```

Union Operator (∪) Example

Table 1: COURSE

```
Course_Id  Student_Name   Student_Id
---------  ------------   ----------
C101        Aditya         S901
C104        Aditya         S901
C106        Steve          S911
C109        Paul           S921
C115        Lucy           S931
```

Table 2: STUDENT

```
Student_Id     Student_Name   Student_Age
------------   ----------     -----------
S901           Aditya         19
S911           Steve          18
S921           Paul           19
S931           Lucy           17
S941           Carl           16
S951           Rick           18
```

Query:

```
∏ Student_Name (COURSE) ∪ ∏ Student_Name (STUDENT)
```

Output:

```
Student_Name
------------
Aditya
Carl
Paul
Lucy
Rick
Steve
```

Note: As you can see there are no duplicate names present in the output even though we had few common names in both the tables, also in the COURSE table we had the duplicate name itself.

Intersection Operator (∩)

Intersection operator is denoted by ∩ symbol and it is used to select common rows (tuples) from two tables (relations).

Lets say we have two relations R1 and R2 both have same columns and we want to select all those tuples(rows) that are present in both the relations, then in that case we can apply intersection operation on these two relations R1 ∩ R2.

Note: Only those rows that are present in both the tables will appear in the result set.

Syntax of Intersection Operator (∩)

```
table_name1 ∩ table_name2
```

Intersection Operator (∩) Example

Lets take the same example that we have taken above.
Table 1: COURSE

```
Course_Id  Student_Name   Student_Id
---------  ------------   ----------
C101        Aditya         S901
C104        Aditya         S901
C106        Steve          S911
C109        Paul           S921
C115        Lucy           S931
```

Table 2: STUDENT

```
Student_Id     Student_Name   Student_Age
------------   ----------     -----------
S901           Aditya         19
S911           Steve          18
S921           Paul           19
S931           Lucy           17
S941           Carl           16
S951           Rick           18
```

Query:

```
∏ Student_Name (COURSE) ∩ ∏ Student_Name (STUDENT)
```

Output:

```
Student_Name
------------
Aditya
Steve
Paul
Lucy
```

Set Difference (-)

Set Difference is denoted by – symbol. Lets say we have two relations R1 and R2 and we want to select all those tuples(rows) that are present in Relation

R1 but **not** present in Relation R2, this can be done using Set difference R1 – R2.

Syntax of Set Difference (-)

```
table_name1 - table_name2
```

Set Difference (-) Example

Lets take the same tables COURSE and STUDENT that we have seen above.

Query:
Lets write a query to select those student names that are present in STUDENT table but not present in COURSE table.

```
∏ Student_Name (STUDENT) - ∏ Student_Name (COURSE)
```

Output:

```
Student_Name
------------
Carl
Rick
```

Cartesian product (X)

Cartesian Product is denoted by X symbol. Lets say we have two relations R1 and R2 then the cartesian product of these two relations (R1 X R2) would combine each tuple of first relation R1 with the each tuple of second relation R2. I know it sounds confusing but once we take an example of this, you will be able to understand this.

Syntax of Cartesian product (X)

```
R1 X R2
```

Cartesian product (X) Example

Table 1: R

```
Col_A    Col_B
-----    ------
AA        100
BB        200
CC        300
```

Table 2: S

```
Col_X     Col_Y
-----     -----
XX         99
```

```
YY          11
ZZ          101
```

Query:
Lets find the cartesian product of table R and S.

```
R X S
```

Output:

```
Col_A    Col_B     Col_X     Col_Y
-----    ------    ------    ------
AA        100      XX         99
AA        100      YY         11
AA        100      ZZ         101
BB        200      XX         99
BB        200      YY         11
BB        200      ZZ         101
CC        300      XX         99
CC        300      YY         11
CC        300      ZZ         101
```

Note: The number of rows in the output will always be the cross product of number of rows in each table. In our example table 1 has 3 rows and table 2 has 3 rows so the output has 3×3 = 9 rows.

Joins in DBMS

What is Join in DBMS?

Join in DBMS is a binary operation which allows you to combine join product and selection in one single statement. The goal of creating a join condition is that it helps you to combine the data from two or more DBMS tables. The tables in DBMS are associated using the primary key and foreign keys.

In this DBMS tutorial, you will learn:

- Types of Join
- Inner Join
  - Theta Join
  - EQUI join:
  - Natural Join (⋈)
- Outer Join
  - Left Outer Join (A   B)
  - Right Outer Join (A ⟖ B)

- Full Outer Join (A ⟗ B)

Types of Join

There are mainly two types of joins in DBMS:

1. Inner Joins: Theta, Natural, EQUI
2. Outer Join: Left, Right, Full

Let's see them in detail:

Inner Join

INNER JOIN is used to return rows from both tables which satisfy the given condition. It is the most widely used join operation and can be considered as a default join-type

An Inner join or equijoin is a comparator-based join which uses equality comparisons in the join-predicate. However, if you use other comparison operators like ">" it can't be called equijoin.

Inner Join further divided into three subtypes:

- Theta join
- Natural join
- EQUI join

Theta Join

THETA JOIN allows you to merge two tables based on the condition represented by theta. Theta joins work for all comparison operators. It is denoted by symbol **θ**. The general case of JOIN operation is called a Theta join.

Syntax:

```
A  ⋈θ  B
```

Theta join can use any conditions in the selection criteria.

Consider the following tables.

Table A		Table B	
column 1	column 2	column 1	column 2
1	1	1	1
1	2	1	3

For example:

$A \bowtie_{A.column\ 2\ >\ B.column\ 2} (B)$

A ⋈ A.column 2 > B.column 2 (B)	
column 1	column 2
1	2

EQUI Join

EQUI JOIN is done when a Theta join uses only the equivalence condition. EQUI join is the most difficult operation to implement efficiently in an RDBMS, and one reason why RDBMS have essential performance problems.

For example:

$A \bowtie_{A.column\ 2\ =\ B.column\ 2} (B)$

A ⋈ A.column 2 = B.column 2 (B)	
column 1	column 2

1	1
1	2

Natural Join (⋈)

NATURAL JOIN does not utilize any of the comparison operators. In this type of join, the attributes should have the same name and domain. In Natural Join, there should be at least one common attribute between two relations.

It performs selection forming equality on those attributes which appear in both relations and eliminates the duplicate attributes.

Example:

Consider the following two tables

C	
Num	**Square**
2	4
3	9

D	
Num	**Cube**
2	8
3	18

```
C ⋈ D
```

C ⋈ D		
Num	**Square**	**Cube**
2	4	8
3	9	18

Outer Join

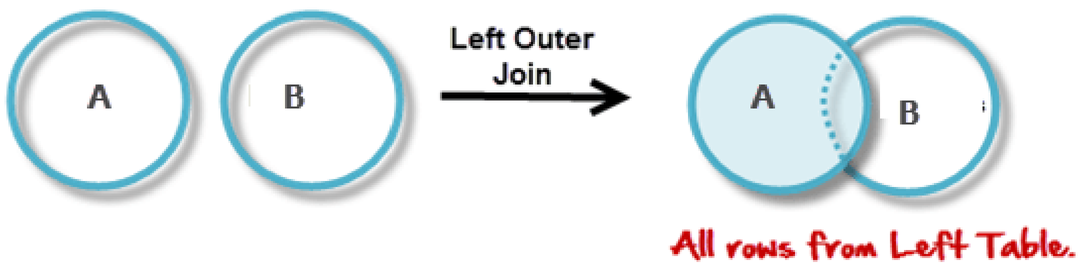
An **OUTER JOIN** doesn't require each record in the two join tables to have a matching record. In this type of join, the table retains each record even if no other matching record exists.

Three types of Outer Joins are:

- Left Outer Join
- Right Outer Join
- Full Outer Join

Left Outer Join (A ⟕ B)

LEFT JOIN returns all the rows from the table on the left even if no matching rows have been found in the table on the right. When no matching record found in the table on the right, NULL is returned.



Consider the following 2 Tables

A

Num	Square
2	4
3	9
4	16

B	
Num	**Cube**
2	8
3	18
5	75

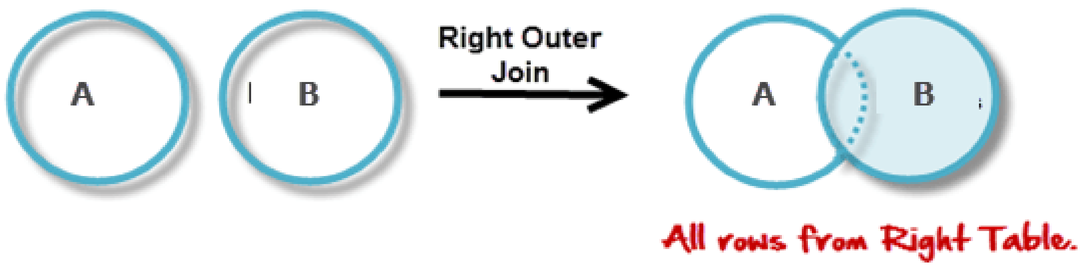
A ⟕ B

A ⋈ B		
Num	**Square**	**Cube**
2	4	8
3	9	18
4	16	-

Right Outer Join (A ⟖ B)

RIGHT JOIN returns all the columns from the table on the right even if no matching rows have been found in the table on the left. Where no matches have been found in the table on the left, NULL is returned. RIGHT outer JOIN is the opposite of LEFT JOIN

In our example, let's assume that you need to get the names of members and movies rented by them. Now we have a new member who has not rented any movie yet.



```
A ⟖ B
```

A ⋈ B		
Num	**Cube**	**Square**
2	8	4
3	18	9
5	75	-

Full Outer Join (A ⟗ B)

In a **FULL OUTER JOIN** , all tuples from both relations are included in the result, irrespective of the matching condition.

Example:

```
A ⟗ B
```

A ⋈ B		
Num	**Square**	**Cube**
2	4	8
3	9	18
4	16	-
5	-	75

Summary:

- There are mainly two types of joins in DBMS 1) Inner Join 2) Outer Join
- An inner join is the widely used join operation and can be considered as a default join-type.
- Inner Join is further divided into three subtypes: 1) Theta join 2) Natural join 3) EQUI join
- Theta Join allows you to merge two tables based on the condition represented by theta
- When a theta join uses only equivalence condition, it becomes an equi join.
- Natural join does not utilize any of the comparison operators.
- An outer join doesn't require each record in the two join tables to have a matching record.
- Outer Join is further divided into three subtypes are: 1)Left Outer Join 2) Right Outer Join 3) Full Outer Join
- The LEFT Outer Join returns all the rows from the table on the left, even if no matching rows have been found in the table on the right.
- The RIGHT Outer Join returns all the columns from the table on the right, even if no matching rows have been found in the table on the left.
- In a full outer join, all tuples from both relations are included in the result, irrespective of the matching condition.

What is Self Join in SQL?

A self join is a join in which a table is joined with itself (which is also called Unary relationships), especially when the table has a FOREIGN KEY which references its own PRIMARY KEY. To join a table itself means that each row of the table is combined with itself and with every other row of the table.

The self join can be viewed as a join of two copies of the same table. The table is not actually copied, but SQL performs the command as though it were.

The syntax of the command for joining a table to itself is almost same as that for joining two different tables. To distinguish the column names from one another, aliases for the actual the table name are used, since both the tables have the same name. Table name aliases are defined in the FROM clause of the SELECT statement. See the syntax :

```
SELECT a.column_name, b.column_name...
FROM table1 a, table1 b
WHERE a.common_filed = b.common_field;
```

For this tutorial we have used a table EMPLOYEE, that has one-to-many relationship.

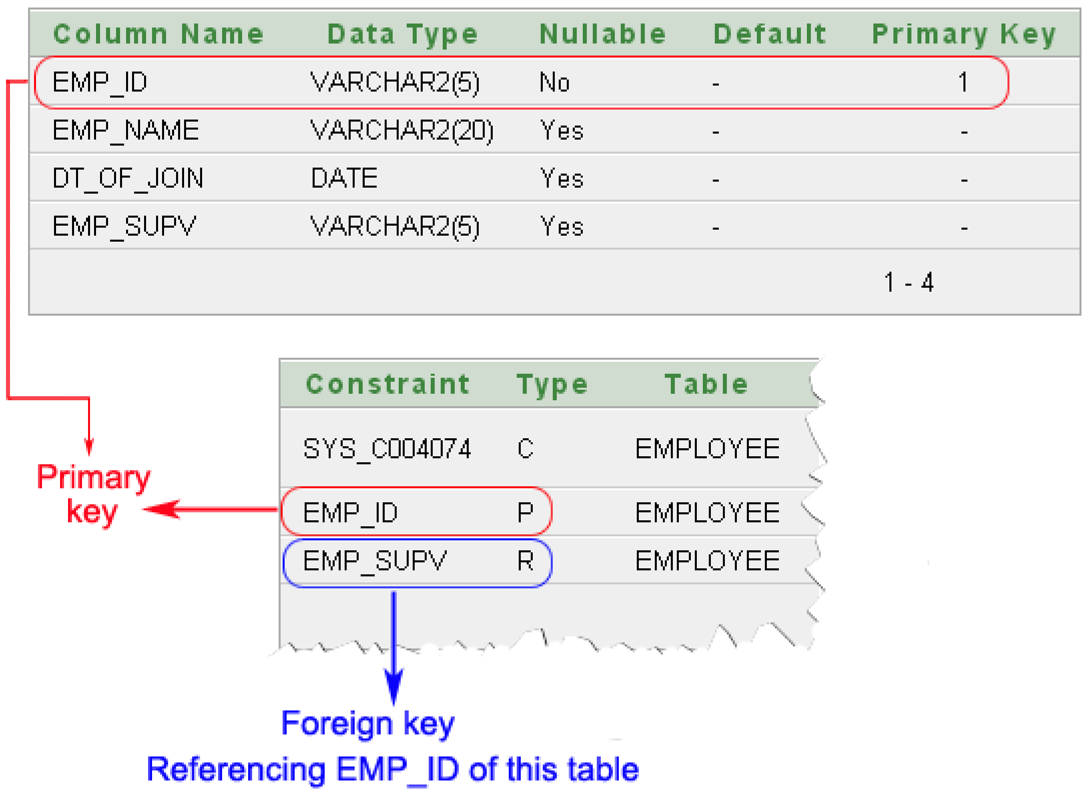
Code to create the table EMPLOYEE

SQL Code:

```
CREATE TABLE employee(emp_id varchar(5) NOT NULL,

emp_name varchar(20) NULL,

dt_of_join date NULL,

emp_supv varchar(5) NULL,

CONSTRAINT emp_id PRIMARY KEY(emp_id) ,

CONSTRAINT emp_supv FOREIGN KEY(emp_supv)

REFERENCESemployee(emp_id));
```

Copy

The structure of the table



Column Name	Data Type	Nullable	Default	Primary Key
EMP_ID	VARCHAR2(5)	No	-	1
EMP_NAME	VARCHAR2(20)	Yes	-	-
DT_OF_JOIN	DATE	Yes	-	-
EMP_SUPV	VARCHAR2(5)	Yes	-	-
				1 - 4

Constraint	Type	Table
SYS_C004074	C	EMPLOYEE
EMP_ID	P	EMPLOYEE
EMP_SUPV	R	EMPLOYEE

In the EMPLOYEE table displayed above, emp_id is the primary key. emp_supv is the foreign key (this is the supervisor's employee id).

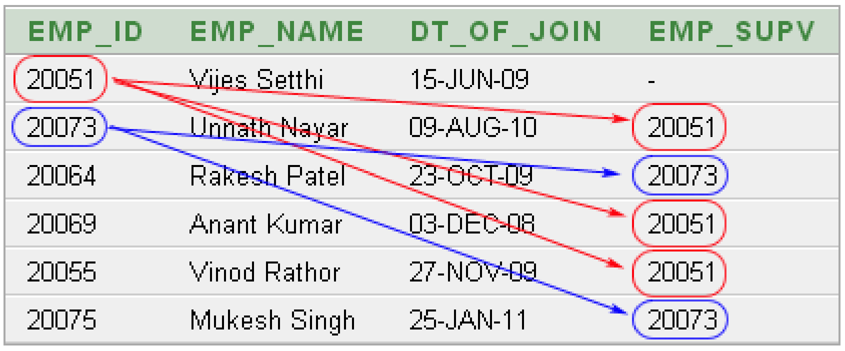
If we want a list of employees and the names of their supervisors, we'll have to JOIN the EMPLOYEE table to itself to get this list.

Unary relationship to employee

How the employees are related to themselves:

- An employee may report to another employee (supervisor).
- An employee may supervise himself (i.e. zero) to many employees (subordinates).

We have the following data into the table EMPLOYEE.



EMP_ID	EMP_NAME	DT_OF_JOIN	EMP_SUPV
20051	Vijes Setthi	15-JUN-09	-
20073	Unnath Nayar	09-AUG-10	20051
20064	Rakesh Patel	23-OCT-09	20073
20069	Anant Kumar	03-DEC-08	20051
20055	Vinod Rathor	27-NOV-09	20051
20075	Mukesh Singh	25-JAN-11	20073

The above data shows:

- Unnath Nayar's supervisor is Vijes Setthi
- Anant Kumar and Vinod Rathor can also report to Vijes Setthi.
- Rakesh Patel and Mukesh Singh are under supervison of Unnith Nayar.

Example of SQL SELF JOIN

In the following example, we will use the table EMPLOYEE twice and in order to do this we will use the alias of the table.

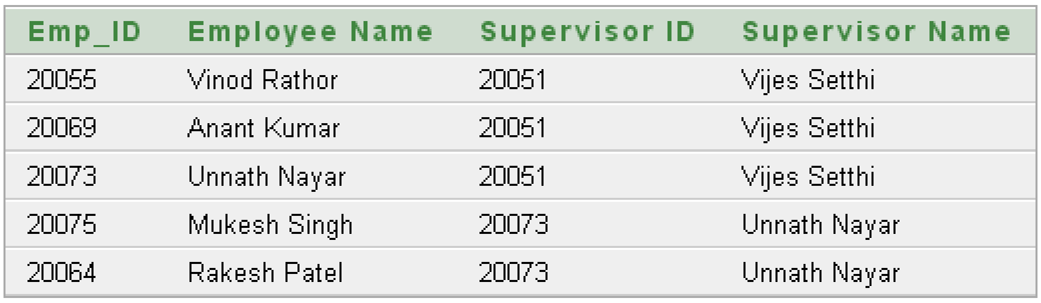
To get the list of employees and their supervisor the following SQL statement has used:

SQL Code:

```
SELECT a.emp_id AS "Emp_ID",a.emp_name AS "Employee Name",

b.emp_id AS "Supervisor ID",b.emp_name AS "Supervisor Name"

FROM employee a, employee b

WHERE a.emp_supv = b.emp_id;
```

Copy

Output:



Emp_ID	Employee Name	Supervisor ID	Supervisor Name
20055	Vinod Rathor	20051	Vijes Setthi
20069	Anant Kumar	20051	Vijes Setthi
20073	Unnath Nayar	20051	Vijes Setthi
20075	Mukesh Singh	20073	Unnath Nayar
20064	Rakesh Patel	20073	Unnath Nayar

Division Operator in SQL

The division operator is used when we have to evaluate queries which contain the keyword `ALL`.

Some instances where division operator is used are:

1. Which person has account in all the banks of a particular city?
2. Which students have taken all the courses required to graduate?

In above specified problem statements, the description after the keyword `'all'` defines a set which contains some elements and the final result contains those units which satisfy these requirements.

Another way how you can identify the usage of division operator is by using the logical implication of `if...then`. In context of the above two examples, we can see that the queries mean that,

1. If there is a bank in that particular city, that person must have an account in that bank.
2. If there is a course in the list of courses required to be graduated, that person must have taken that course.

Do not worry if you are not clear with all this new things right away, we will try to expain as we move on with this tutorial.

We shall see the second example, mentioned above, in detail.

Table 1: Course_Taken → It consists of the names of Students against the courses that they have taken.

Student_Name	Course
Robert	Databases
Robert	Programming Languages
David	Databases
David	Operating Systems
Hannah	Programming Languages
Hannah	Machine Learning
Tom	Operating Systems

Table 2: Course_Required → It consists of the courses that one is required to take in order to graduate.

Course
Databases
Programming Languages

Using Division Operator

So now, let's try to find out the correct SQL query for getting results for the first requirement, which is:

Query: Find all the students who can graduate. (i.e. who have taken all the subjects required for one to graduate.)

Unfortunately, there is no direct way by which we can express the division operator. Let's walk through the steps, to write the query for the division operator.

1. Find all the students

Create a set of all students that have taken courses. This can be done easily using the following command.

```
CREATE TABLE AllStudents AS SELECT DISTINCT Student_Name FROM Course_Taken
```

This command will return the table **AllStudents**, as the resultset:

Student_name
Robert
David
Hannah
Tom

2. Find all the students and the courses required to graduate

Next, we will create a set of students and the courses they need to graduate. We can express this in the form of Cartesian Product of **AllStudents** and **Course_Required** using the following command.

```
CREATE table StudentsAndRequired AS

SELECT AllStudents.Student_Name, Course_Required.Course

FROM AllStudents, Course_Required
```

Now the new resultset - table **StudentsAndRequired** will be:

Student_Name	**Course**
Robert	Databases
Robert	Programming Languages
David	Databases
David	Programming Languages
Hannah	Databases
Hannah	Programming Languages
Tom	Databases

Tom	Programming Languages

3. Find all the students and the required courses they have not taken

Here, we are taking our first step for finding the students who cannot graduate. The idea is to simply find the students who have not taken certain courses that are required for graduation and hence they wont be able to graduate. This is simply all those tuples/rows which are present in **StudentsAndRequired** and not present in **Course_Taken**.

```
CREATE  table StudentsAndNotTaken AS

SELECT * FROM StudentsAndRequired WHERE NOT EXISTS

(Select * FROM Course_Taken WHERE StudentsAndRequired.Student_Name =
Course_Taken.Student_Name

AND StudentsAndRequired.Course = Course_Taken.Course)
```

The table **StudentsAndNotTaken** comes out to be:

Student_Name	**Course**
David	Programming Languages
Hannah	Databases
Tom	Databases
Tom	Programming Languages

4. Find all students who cannot graduate

All the students who are present in the table **StudentsAndNotTaken** are the ones who cannot graduate. Therefore, we can find the students who cannot graduate as,

```
CREATE table CannotGraduate AS SELECT DISTINCT Student_Name FROM
StudentsAndNotTaken
```

Student_name
David
Hannah
Tom

5. Find all students who can graduate

The students who can graduate are simply those who are present in **AllStudents** but not in **CannotGraduate**. This can be done by the following query:

```
CREATE Table CanGraduate AS SELECT * FROM AllStudents

WHERE NOT EXISTS

(SELECT * FROM CannotGraduate WHERE

    CannotGraduate.Student_name = AllStudents.Student_name)
```

The results will be as follows:

Student_name

Robert

Hence we just learned, how different steps can lead us to the final answer. Now let us see how to write all these 5 steps in one single query so that we do not have to create so many tables.

```
SELECT DISTINCT  x.Student_Name FROM Course_Taken AS x WHERE NOT

EXISTS(SELECT * FROM Course_Required AS y WHERE NOT

EXISTS(SELECT * FROM Course_Taken AS z

    WHERE z.Student_name = x.Student_name

    AND z.Course = y.Course ))
```

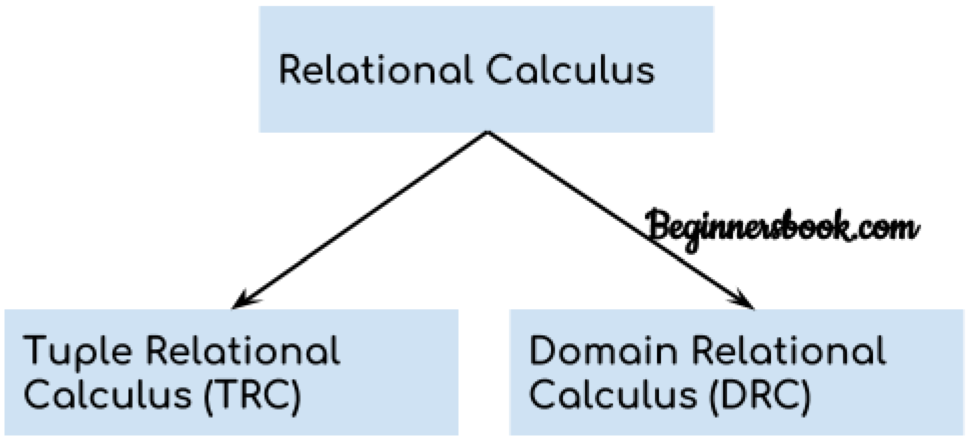
Student_name
Robert

This gives us the same result just like the 5 steps above.

What is Relational Calculus?

Relational calculus is a non-procedural query language that tells the system what data to be retrieved but doesn’t tell how to retrieve it.

Types of Relational Calculus


Relational Calculus
Beginnersbook.com
Tuple Relational Calculus (TRC)
Domain Relational Calculus (DRC)

1. Tuple Relational Calculus (TRC)

Tuple relational calculus is used for selecting those tuples that satisfy the given condition.
Table: Student

```
First_Name     Last_Name     Age
----------     ---------     ----
Ajeet          Singh         30
Chaitanya      Singh         31
Rajeev         Bhatia        27
Carl           Pratap        28
```

Lets write relational calculus queries.

Query to display the last name of those students where age is greater than 30

```
{ t.Last_Name | Student(t) AND t.age > 30 }
```

In the above query you can see two parts separated by | symbol. The second part is where we define the condition and in the first part we specify the fields which we want to display for the selected tuples.

The result of the above query would be:

```
Last_Name
---------
Singh
```

Query to display all the details of students where Last name is ‘Singh’

```
{ t | Student(t) AND t.Last_Name = 'Singh' }
```

Output:

```
First_Name     Last_Name     Age
----------     ---------     ----
Ajeet          Singh         30
Chaitanya      Singh         31
```

2. Domain Relational Calculus (DRC)

In domain relational calculus the records are filtered based on the domains.
Again we take the same table to understand how DRC works.
Table: Student

```
First_Name     Last_Name     Age
----------     ---------     ----
Ajeet          Singh         30
Chaitanya      Singh         31
Rajeev         Bhatia        27
Carl           Pratap        28
```

Query to find the first name and age of students where student age is greater than 27

```
{< First_Name, Age > | ∈ Student ∧ Age > 27}
```

Note:
The symbols used for logical operators are: ∧ for AND, ∨ for OR and ┐ for NOT.

Output:

```
First_Name     Age
----------     ----
Ajeet          30
Chaitanya      31
Carl           28
```