**Primary key in SQL**

* A table consists of columns and rows. Typically, a table has a column or set of columns whose values uniquely identify each row in the table. This column or the set of columns is called the **primary key**.
* The primary key that consists of two or more columns is also known as the **composite primary key**.
* Each table has **one and only one** primary key. The primary key does not accept NULL or duplicate values.

**Creating table with primary key**

CREATE TABLE projects (

project\_id INT PRIMARY KEY,

project\_name VARCHAR(255),

start\_date DATE NOT NULL,

end\_date DATE NOT NULL

);

CREATE TABLE project\_assignments (

project\_id INT,

employee\_id INT,

join\_date DATE NOT NULL,

CONSTRAINT pk\_assgn PRIMARY KEY (project\_id)

);

CREATE TABLE project\_assignments (

project\_id INT,

employee\_id INT,

join\_date DATE NOT NULL,

CONSTRAINT pk\_assgn PRIMARY KEY (project\_id, employee\_id)

);

**Adding the primary key with ALTER TABLE statement**

CREATE TABLE project\_milestones(

milestone\_id INT,

project\_id INT,

milestone\_name VARCHAR(100)

);

ALTER TABLE project\_milestones

ADD CONSTRAINT pk\_milestone\_id PRIMARY KEY (milestone\_id);

ALTER TABLE project\_milestones

ADD PRIMARY KEY (milestone\_id);

**Removing the primary key constraint**

ALTER TABLE table\_name

DROP CONSTRAINT primary\_key\_constraint;

ALTER TABLE table\_name

DROP PRIMARY KEY;

ALTER TABLE project\_milestones

DROP CONSTRAINT pk\_milestone\_id;

**SQL foreign key constraint**

* A foreign key is a column or a group of columns that enforces a link between the data in two tables. In a foreign key reference, the primary key column (or columns) of the second table. The column (or columns) of the second table becomes the foreign key.
* You use the FOREIGN KEY constraint to create a foreign key when you create or alter table. Let’s take a simple example to get a better understanding.

**SQL FOREIGN KEY constraint examples**

CREATE TABLE projects (

project\_id INT AUTO\_INCREMENT PRIMARY KEY,

project\_name VARCHAR(255),

start\_date DATE NOT NULL,

end\_date DATE NOT NULL

);

CREATE TABLE project\_milestones(

milestone\_id INT AUTO\_INCREMENT PRIMARY KEY,

project\_id INT,

milestone\_name VARCHAR(100)

);

* Each project may have zero or more milestones while one milestone must belong to one and only one project. The application that uses these tables must ensure that for each row in the project\_milestones table there exists the corresponding row in the projects table. In other words, a milestone cannot exist without a project.
* Unfortunately, users may edit the database using client tool or if there is a bug in the application, a row might be added to the project\_milestones table that does not correspond to any row in the projects table. Or user may delete a row in the projects table, leaving orphaned rows in the project\_milestones table. This causes the application not to work properly.
* The solution is to add an SQL FOREIGN KEY constraint to project\_milestones table to enforce the relationship between the projects and project\_milestones tables.
* You can create the FOREIGN KEY constraint when you create the table as follows:

CREATE TABLE project\_milestones (

milestone\_id INT AUTO\_INCREMENT PRIMARY KEY,

project\_id INT,

milestone\_name VARCHAR(100),

FOREIGN KEY (project\_id)

REFERENCES projects (project\_id)

);

* The FOREIGN KEY clause promotes the project\_id of the project\_milestones table to become the foreign key that is referenced to the project\_id of the projects table.

FOREIGN KEY (project\_id)

REFERENCES projects (project\_id)

* You can assign a name to a FOREIGN KEY constraint as follows:

CREATE TABLE project\_milestones (

milestone\_id INT AUTO\_INCREMENT PRIMARY KEY,

project\_id INT,

milestone\_name VARCHAR(100),

CONSTRAINT fk\_project FOREIGN KEY (project\_id)

REFERENCES projects (project\_id)

);

* fk\_project is the name of the FOREIGN KEY constraint.

**Adding FOREIGN KEY constraints to existing tables**

* To add a FOREIGN KEY constraint to existing table, you use the ALTER TABLE statement.

ALTER TABLE table\_1

ADD CONSTRAINT fk\_name FOREIGN KEY (fk\_key\_column)

REFERENCES table\_2(pk\_key\_column)

* Suppose that project\_milestones already exists without any predefined foreign key and you want to define a FOREIGN KEY constraint for the project\_id column. To do so, you use the following ALTER TABLE statement:

ALTER TABLE project\_milestones

ADD CONSTRAINT fk\_project FOREIGN KEY(project\_id)

REFERENCES projects(project\_id);

**Removing foreign key constraints**

* To remove a foreign key constraint, you also use the ALTER TABLE statement as follows:

ALTER TABLE table\_name

DROP CONSTRAINT fk\_name;

* If you are using MySQL, you can use a cleaner syntax as follows:

ALTER TABLE table\_name

DROP FOREIGN KEY fk\_name;

* For example, to remove the fk\_project foreign key constraint, you use the following statement:

ALTER TABLE project\_milestones

DROP CONSTRAINT fk\_project;

**Normalization**

* Normalization is the process of improving the properties of a database by sequentially dividing one table into several tables. The reason for normalization is the presence of anomalies in the unnormalized database. An anomaly is a situation in which there is a complication of data processing and a violation of consistency, i.e., the correctness or adequacy of information storage. In this topic, we will start with a very badly organized database: Children of Employees and work our way together through increasing normalization to a stage where the database is more or less well structured. Stay tuned!

**First normal form**

* Let’s start with the first normal form (1NF). The relationship is in 1NF, when all data in its cells is atomic, i.e. simple and non-separable.

**Second normal form**

* As you can see above, the relation in 1NF contains data duplication and anomalies. An important concept with 2NF is functional dependence. Functional dependence is more of a semantic concept and is denoted as x->y (read as x functionally defines y, or y is functionally dependent on x). If there is such a dependence between attributes, then this means that for any two identical values of x, exactly the same value of y will correspond to them (i.e., x1->y1 and x1->y2, then y1=y2). In this case, x is called the dependency determinant.
* The relation is in 2NF when it is in NF, and all non-key attributes are entirely dependent on the key, but aren’t a part of it. The dependence on the entire composite key is called complete. If a non-key attribute depends only on a portion of a composite key, then the dependency is incomplete.
* In this case, we mean the dependence on the potential key, not the primary one.

**Third normal form**

* Moving on to 3NF, it is necessary to introduce the concept of transitive dependence. A transitive dependence exists when x->y and y->z, which leads to x->z(say x depends transitively on z).
* A relation is in 3NF when it is in 2NF and there is no transitive dependency on non-key attributes.

**Boyce-Codd normal form**

* BCNF appears when it comes to having multiple super keys. In the relational data model a super key is a set of attributes that uniquely identifies each tuple of a relation.
* It is different from the primary key, but only that the primary key is the key chosen by the programmer for implementation as a key, therefore any primary key is also potential.
* A relation is BCNF when it is in 3NF and any determinant is the key of any dependence.

**SQL JOIN Queries**

* JOIN clause is used to combine rows from two or more tables, based on a related column between them.

**INNTER JOIN**

* An INNER JOIN in SQL is a type of join that returns the records with matching values in both tables. This operation compares each row of the first table with each row of the second table to find all pairs of rows that satisfy the join predicate.
* Few things to consider in case of INNER JOIN:
  + It is a default join in SQL. If you mention JOIN in your query without specifying the type, SQL considers it as an INNER JOIN.
  + It returns only the matching rows from both the tables.
  + If there is no match, the returned is an empty result.
* Syntax:

SELECT column\_name(s)

FROM table1

INNER JOIN table2

ON table1.column\_name = table2.column\_name;

**LEFT JOIN**

* The SQL LEFT JOIN combines rows from two or more tables based on a related column between them and returns all rows from the left table(table1) and the matched rows from the right table(table2). If there is no match, the result is NULL on the right side.
* Syntax:

SELECT column\_name(s)

FROM table1

LEFT JOIN table2

ON table1.column\_name = table2.column\_name;

* How SQL LEFT JOIN Works: The LEFT JOIN keyword returns all records from the left table(table), and the matched records from the right table(table2). The result is NULL from the right side, if there is no match.

**RIGHT JOIN**

* The RIGHT JOIN keyword returns all records from the right table(table2), and the matched records from the left table(table1). If there is no match, the result is NULL on the left side.
* Syntax:

SELECT column\_name(s)

FROM table1

RIGHT JOIN table2

ON table1.column\_name = table2\_column\_name;

**FULL OUTER JOIN**

* A FULL OUTER JOIN in SQL is a method to combine rows from two or more tables, based on a related column between them. It returns all rows from the left table(table1) and from the right table(table2).
* The FULL OUTER JOIN keyword combines the results of both left and right outer joins and returns all (matched or unmatched) rows from the tables on both sides of the join clause.
* Syntax:

SELECT column\_name(s)

FROM table1

FULL OUTER JOIN table2

ON table1.column\_name = table2.column\_name;

**Self Join**

* A SELF JOIN is a standard SQL operation where a table is joined to itself. This might sound counter-intuitive, but it’s actually quite useful in scenarios where comparison operations need to be made within a table. Essentially, it is used to combine rows with other rows in the same table when there’s a match based on the condition provided.
* It’s important to note that since it’s a join operation on the same table, alias(es) for table(s) must be used to avoid confusion during the join operation.
* Syntax:

SELECT a.column\_name, b.column\_name

FROM table\_name AS a, table\_name AS b

WHERE a.common\_field = b.common\_field;

* table\_name: is the name of the table to join to itself.
* a and b: are different aliases for the same table.
* column\_name: specify the columns that should be returned as a result of the SQL SELF JOIN statement.
* WHERE a.common\_field = b.common\_field: is the condition for the join.

**Cross Join**

* The cross join in SQL is used to combine every row of the first table with every row of the second table. It’s also known as the Cartesian product of the two tables. The most important aspect of performing a cross join is that it does not require any condition to join.
* The issue with cross join is it returns the Cartesian product of the two tables, which can result in large numbers of rows and heavy resource usage. It’s hence critical to use them wisely and only when necessary.
* Syntax
* Here’s the generic syntax for implementing a CROSS JOIN:

SELECT column\_name(s)

FROM table1

CROSS JOIN table2;

* You can alternatively use the below syntax to achieve the same result:

SELECT column\_name(s)

FROM table1, table2;

* Both syntax return the Cartesian product of table1 and table2.

**GROUP BY**

* “Group By” is a clause in SQL that is used to arrange identical data into groups. This clause comes under the category of Group Functions, alongside the likes of Count, Sum, Average, etc.
* Syntax:

SELECT column\_name [, list\_of\_other\_columns]

, aggregation [, list\_of\_aggregations]

FROM table\_name

[WHERE list\_of\_conditions]

GROUP BY column\_name [, list\_of\_other\_columns]

[HAVING list\_of\_aggregate\_conditions]

[ORDER BY list\_of\_columns/aliases];

**ORDER BY**

* The ORDER BY clause in SQL is used to sort the result-set from a SELECT statement in ascending or descending order. It sorts the records in ascending order by default. If you want to sort the records in descending order, you have to use the DESC keyword.
* Syntax:

ORDER BY expr1 [ASC, DESC], ..., exprN [ASC, DESC];

UPDATE

* The UPDATE command in SQL is used to modify the existing records in a table. This command is useful when you need to update existing data within a database.
* Here are important points to remember before updating records in SQL:
* The WHERE clause in the UPDATE statement specifies which records to modify. If you omit the WHERE clause, all records in the table will be updated!
* Be careful when updating records in SQL. If you inadvertently run an UPDATE statement without a WHERE clause, you will rewrite all the data in the table.
* Syntax:

UPDATE table\_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

* In this syntax:
  + table\_name: Specifies the table where you want to update records.
  + SET: This keyword is used to set the column values.
  + column1, column2... columnN: These are the columns of the table that you want to change.
  + value1, value2... valueN: These are the new values that you want to assign for your columns.
  + WHERE: This clause specifies which records need to be updated. It selects records based on one or more conditions.