SQL Statements

SELECT **DISTINCT** column1, column2, ...
FROM table_name;

The following SQL statement selects only the DISTINCT values from the "Country" column in the "Customers" table.

SELECT DISTINCT Country FROM Customers;

The following SQL statement lists the number of different (distinct) customer countries

SELECT COUNT(DISTINCT Country) FROM Customers;

SQL Statements

The SQL **ORDER BY** Keyword

The ORDER BY keyword is used to sort the result-set in ascending or descending order.

The ORDER BY keyword sorts the records in ascending order by default. To sort the records in descending order, use the DESC keyword.

```
SELECT column1, column2, ...
FROM table_name
ORDER BY column1, column2, ... ASC|DESC;
```

SQL Statements

The SQL **ORDER BY** Keyword

The following SQL statement selects all customers from the "Customers" table, sorted by the "Country" column.

SELECT * FROM Customers ORDER BY Country;

SQL Statements

SQL **NULL** Values

A field with a NULL value is a field with no value.

If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

SQL Statements

SQL **NULL** Values

```
SELECT column_names
FROM table_name
WHERE column_name IS NULL;
```

```
SELECT column_names
FROM table_name
WHERE column_name IS NOT NULL;
```

SQL Statements

SQL **UPDATE** Statement

The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

SQL Statements

SQL **UPDATE** Statement

The UPDATE statement is used to modify the existing records in a table.

```
UPDATE table_name
SET column1 = value1, column2 = value2, ...
WHERE condition;
```

```
UPDATE Customers
SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'
WHERE CustomerID = 1;
```

SQL Statements

SQL **UPDATE** Statement

UPDATE Multiple Records

```
UPDATE Customers
SET ContactName=‘Ram'
WHERE Country=‘USA’;
```

Update Warning!

Be careful when updating records. If you omit the WHERE clause, ALL records will be updated!

SQL Statements

SQL **MIN() and MAX()** Functions

The MIN() function returns the smallest value of the selected column.
The MAX() function returns the largest value of the selected column.

```
SELECT MIN(column_name)
FROM table_name
WHERE condition;
```

```
SELECT MAX(column_name)
FROM table_name
WHERE condition;
```

SQL Statements

SQL **MIN() and MAX()** Functions

```
SELECT MIN(Price) AS SmallestPrice
FROM Products;
```

```
SELECT MAX(Price) AS LargestPrice
FROM Products;
```

SQL Statements

SQL **COUNT(), AVG() and SUM()** Functions

The COUNT() function returns the number of rows that matches a specified criterion.

```
SELECT COUNT(column_name)
FROM table_name
WHERE condition;
```

The AVG() function returns the average value of a numeric column.

```
SELECT AVG(column_name)
FROM table_name
WHERE condition;
```

SQL Statements

SQL **COUNT(), AVG() and SUM()** Functions

The SUM() function returns the total sum of a numeric column.

```
SELECT SUM(column_name)
FROM table_name
WHERE condition;
```

Note: NULL values are not counted/are ignored.

SQL Statements

SQL **Aliases**

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of that query.

An alias is created with the AS keyword.

SQL Statements

SQL **Aliases**

Alias Column Syntax

```
SELECT column_name AS alias_name
FROM table_name;
```

```
SELECT CustomerID AS ID, CustomerName AS Customer
FROM Customers;
```

SQL Statements

SQL **Aliases**

Alias Table Syntax

```
SELECT column_name(s)
FROM table_name AS alias_name;
```

```
SELECT o.OrderID, o.OrderDate, c.CustomerName
FROM Customers AS c, Orders AS o
WHERE c.CustomerName='Around the Horn' AND
c.CustomerID=o.CustomerID;
```

```
SELECT Orders.OrderID, Orders.OrderDate, Customers.CustomerName
FROM Customers, Orders
WHERE Customers.CustomerName='Around the Horn' AND
Customers.CustomerID=Orders.CustomerID;
```

SQL Statements

The SQL **GROUP BY** Statement

The GROUP BY statement groups rows that have the same values into summary rows, like "find the number of customers in each country".

The GROUP BY statement is often used with aggregate functions (COUNT(), MAX(), MIN(), SUM(), AVG()) to group the result-set by one or more columns.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
ORDER BY column_name(s);
```

SQL Statements

The SQL GROUP BY Statement

The following SQL statement lists the number of customers in each country:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

The following SQL statement lists the number of customers in each country, sorted high to low:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
ORDER BY COUNT(CustomerID) DESC;
```

SQL Statements

The SQL **HAVING Clause**

The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
HAVING condition
ORDER BY column_name(s);
```

SQL Statements

The SQL **HAVING Clause**

The following SQL statement lists the number of customers in each country. Only include countries with more than 5 customers:

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5;
```

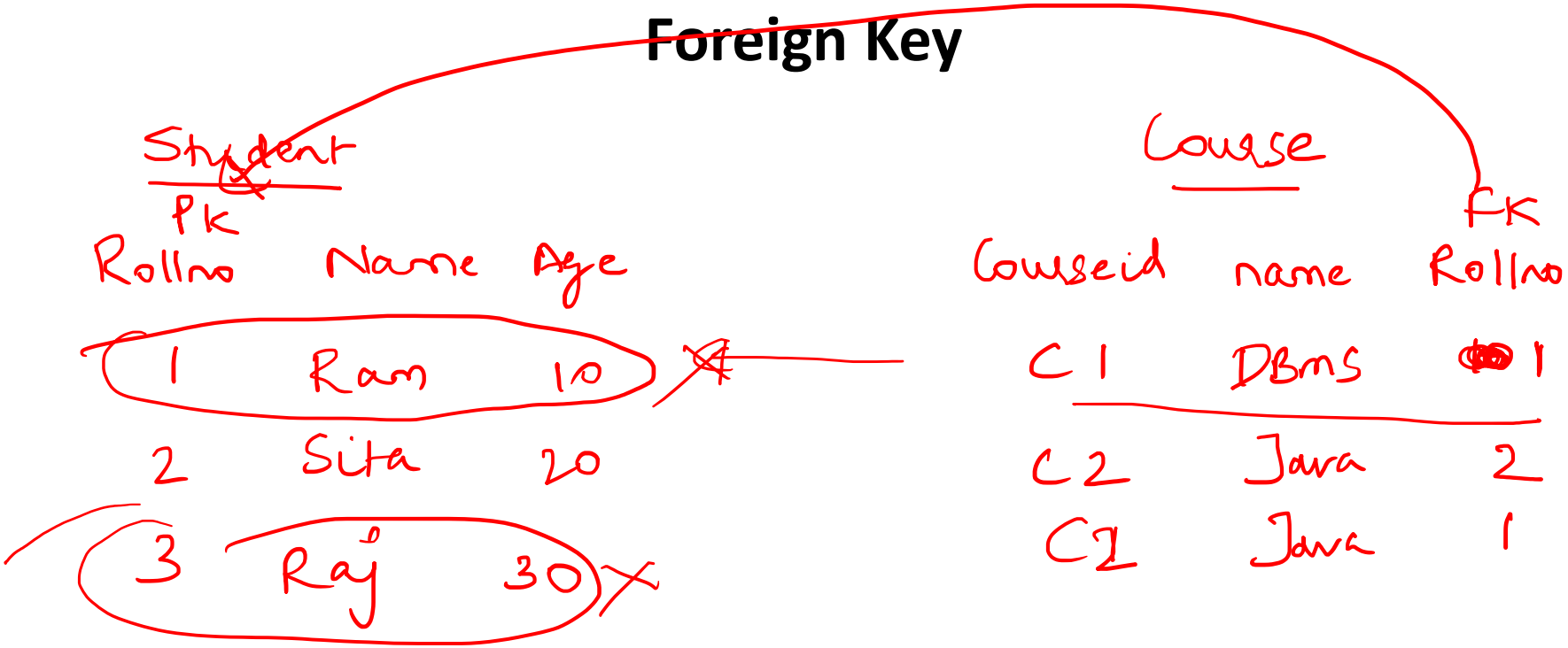
The following SQL statement lists the number of customers in each country, sorted high to low (Only include countries with more than 5 customers):

```
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country
HAVING COUNT(CustomerID) > 5
ORDER BY COUNT(CustomerID) DESC;
```

Foreign Key

- Tables are related to other tables with a primary key foreign key relationship.
- Primary and foreign key relationships are used in relational databases to define many-to-one relationships between tables.
- A FOREIGN KEY is a field (or collection of fields) in one table, that refers to the PRIMARY KEY in another table.
- The table with the foreign key is called the child table(referencing table), and the table with the primary key is called the referenced or parent table.

Foreign Key


Student
PK
Rollno
Name
Age
1
Ram
10
2
Sita
20
3
Raj
30
Course
Courseid
name
Rollno
FK
C1
DBMS
1
C2
Java
2
C1
Java
1

Foreign Key

PERSONS TABLE

PERSONID	FIRSTNAME	LASTNAME	AGE
10	RAM	K	34
20	SITA	T	40
30	RAJU	K	60

ORDERS TABLE

ORDERID	ORDERNUMBER	PERSONID
1	888	20
2	777	10
3	555	20

- Notice that the "PersonID" column in the "Orders" table points to the "PersonID" column in the "Persons" table.
- The "PersonID" column in the "Persons" table is the PRIMARY KEY in the "Persons" table.
- The "PersonID" column in the "Orders" table is a FOREIGN KEY in the "Orders" table.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, because it has to be one of the values contained in the parent table.

Foreign Key

S.NO.	PRIMARY KEY	FOREIGN KEY
1	A primary key is used to ensure data in the specific column is unique.	A foreign key is a column or group of columns in a relational database table that provides a link between data in two tables.
2	It uniquely identifies a record in the relational database table.	It refers to the field in a table which is the primary key of another table.
3	Only one primary key is allowed in a table.	Whereas more than one foreign key are allowed in a table.
4	It is a combination of UNIQUE and Not Null constraints.	It can contain duplicate values and a table in a relational database.
5	It does not allow NULL values.	It can also contain NULL values.
6	Its value cannot be deleted from the parent table.	Its value can be deleted from the child table.

Foreign Key

```
CREATE TABLE Persons (
    PersonID int ,
    Firstname varchar(20),
    Lastname varchar(20),
    Age int,
    PRIMARY KEY (PersonID));

CREATE TABLE Orders (
    OrderID int,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    FOREIGN KEY (PersonID) REFERENCES Persons(PersonID)
);
```

Foreign Key

```
CREATE TABLE Orders (
    OrderID int NOT NULL,
    OrderNumber int NOT NULL,
    PersonID int,
    PRIMARY KEY (OrderID),
    CONSTRAINT FK_PersonOrder FOREIGN KEY (PersonID)
    REFERENCES Persons(PersonID)
);
```

Foreign Key

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES
Persons(PersonID);

ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```

Foreign Key

```
ALTER TABLE Orders
ADD FOREIGN KEY (PersonID) REFERENCES
Persons(PersonID);

ALTER TABLE Orders
ADD CONSTRAINT FK_PersonOrder
FOREIGN KEY (PersonID) REFERENCES Persons(PersonID);

ALTER TABLE Orders
DROP FOREIGN KEY FK_PersonOrder;
```