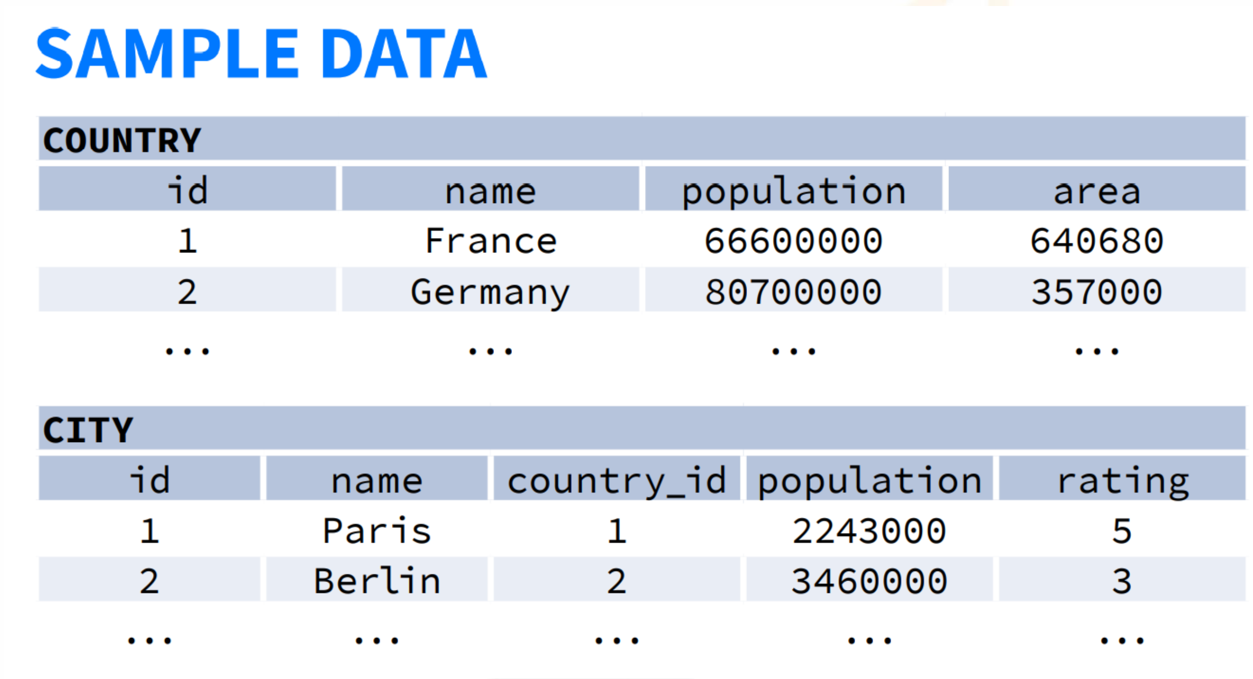
# SQL Basics Cheat Sheet

## SQL

SQL, or Structured Query Language, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.



## SAMPLE DATA

**COUNTRY**

| id | name | population | area |
|---|---|---|---|
| 1 | France | 66600000 | 640680 |
| 2 | Germany | 80700000 | 357000 |
| ... | ... | ... | ... |

**CITY**

| id | name | country_id | population | rating |
|---|---|---|---|---|
| 1 | Paris | 1 | 2243000 | 5 |
| 2 | Berlin | 2 | 3460000 | 3 |
| ... | ... | ... | ... | ... |

## QUERYING SINGLE TABLE

Fetch all columns from the country table:

```
SELECT *
FROM COUNTRY;
```

Using this SELECT statement, the query selects all data from all columns in the COUNTRY'S table.

Fetch id and name columns from the city table:

```
SELECT id, name

FROM city;
```

Fetch city names sorted by the rating column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

```
SELECT name
FROM city
ORDER BY rating DESC;
```

## FILTERING THE OUTPUT

### COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
AND name != 'Madrid';
```

### TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

## OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND 5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

# QUERYING MULTIPLE TABLES

## INNER JOIN

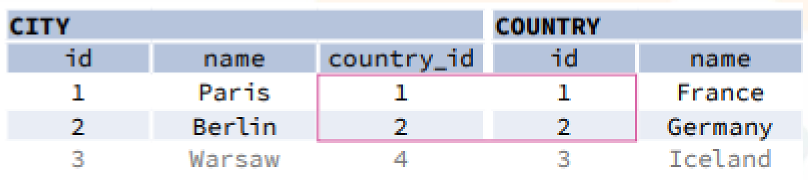
JOIN (or explicitly INNER JOIN) returns rows that have matching values in both tables

SELECT city.name, country.name

FROM city

**[INNER] JOIN** country

ON city.country_id = country.id;



| CITY | | | COUNTRY | |
|---|---|---|---|---|
| id | name | country_id | id | name |
| 1 | Paris | 1 | 1 | France |
| 2 | Berlin | 2 | 2 | Germany |
| 3 | Warsaw | 4 | 3 | Iceland |

## LEFT JOIN

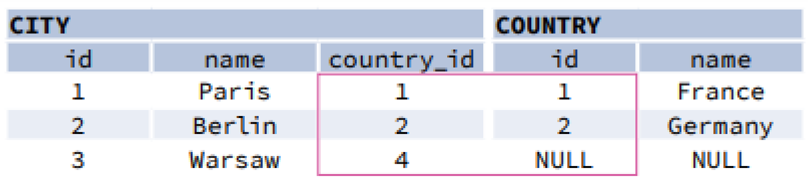
LEFT JOIN returns all rows from the left table with corresponding rows from the right table. If there's no matching row, NULLs are returned as values from the second table.

SELECT city.name, country.name

FROM city

**LEFT JOIN** country

ON city.country_id = country.id;



| CITY | | | COUNTRY | |
|---|---|---|---|---|
| id | name | country_id | id | name |
| 1 | Paris | 1 | 1 | France |
| 2 | Berlin | 2 | 2 | Germany |
| 3 | Warsaw | 4 | NULL | NULL |

## RIGHT JOIN

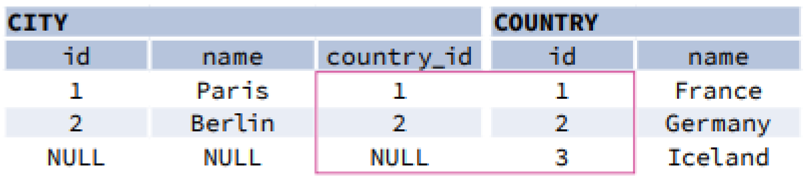
RIGHT JOIN returns all rows from the right table with corresponding rows from the left table. If there's no matching row, NULLs are returned as values from the left table.

SELECT city.name, country.name

 FROM city

**RIGHT JOIN** country

ON city.country_id = country.id;



| CITY | | | COUNTRY | |
|---|---|---|---|---|
| id | name | country_id | id | name |
| 1 | Paris | 1 | 1 | France |
| 2 | Berlin | 2 | 2 | Germany |
| NULL | NULL | NULL | 3 | Iceland |

## FULL JOIN

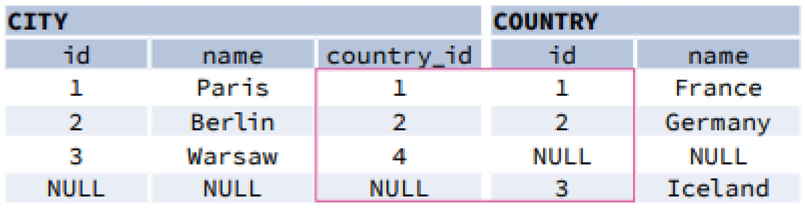
FULL JOIN (or explicitly FULL OUTER JOIN) returns all rows from both tables – if there's no matching row in the second table, NULLs are returned.

SELECT city.name, country.name

FROM city

**FULL [OUTER] JOIN** country

ON city.country_id = country.id;



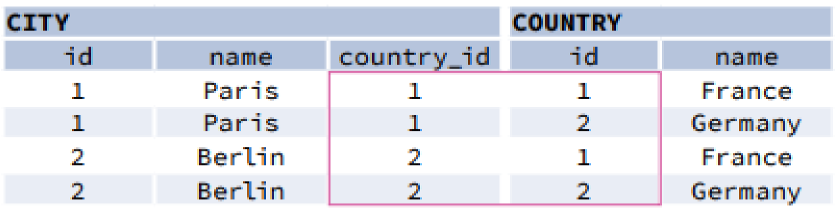
| CITY | | | COUNTRY | |
|---|---|---|---|---|
| id | name | country_id | id | name |
| 1 | Paris | 1 | 1 | France |
| 2 | Berlin | 2 | 2 | Germany |
| 3 | Warsaw | 4 | NULL | NULL |
| NULL | NULL | NULL | 3 | Iceland |

## CROSS JOIN

CROSS JOIN returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;
```

```
SELECT city.name, country.name
FROM city, country;
```

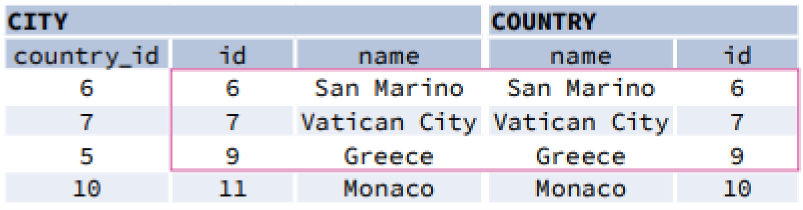


| CITY | | | COUNTRY | |
|---|---|---|---|---|
| id | name | country_id | id | name |
| 1 | Paris | 1 | 1 | France |
| 1 | Paris | 1 | 2 | Germany |
| 2 | Berlin | 2 | 1 | France |
| 2 | Berlin | 2 | 2 | Germany |

## NATURAL JOIN

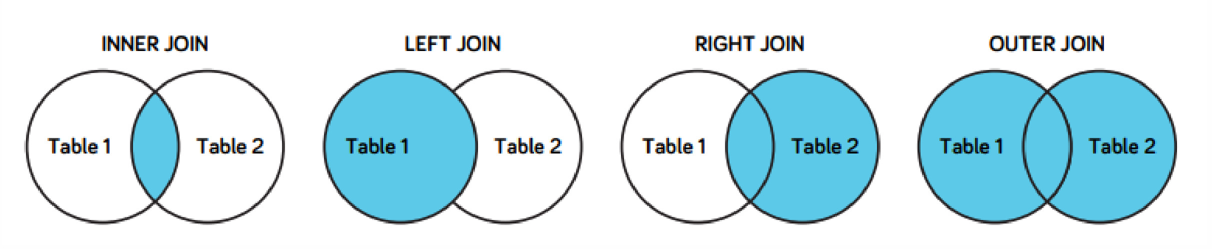
NATURAL JOIN will join tables by all columns with the same name

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```



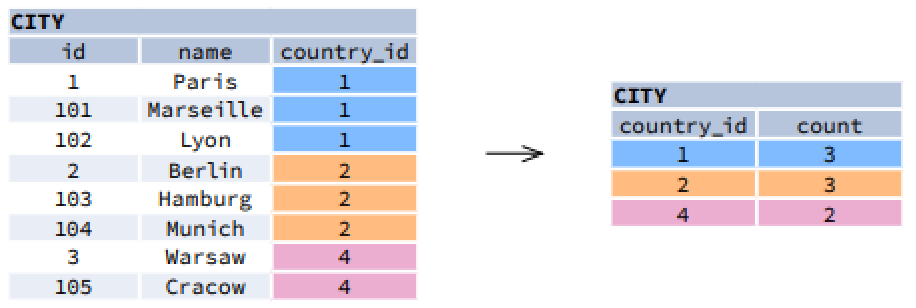
| CITY | | | COUNTRY | |
|---|---|---|---|---|
| country_id | id | name | name | id |
| 6 | 6 | San Marino | San Marino | 6 |
| 7 | 7 | Vatican City | Vatican City | 7 |
| 5 | 9 | Greece | Greece | 9 |
| 10 | 11 | Monaco | Monaco | 10 |

NATURAL JOIN is very rarely used in practice.



## AGGREGATION AND GROUPING

GROUP BY groups together rows that have the same values in specified columns. It computes summaries (aggregates) for each unique combination of values.



CITY

| id | name | country_id |
|---|---|---|
| 1 | Paris | 1 |
| 101 | Marseille | 1 |
| 102 | Lyon | 1 |
| 2 | Berlin | 2 |
| 103 | Hamburg | 2 |
| 104 | Munich | 2 |
| 3 | Warsaw | 4 |
| 105 | Cracow | 4 |

→

CITY

| country_id | count |
|---|---|
| 1 | 3 |
| 2 | 3 |
| 4 | 2 |

### AGGREGATE FUNCTIONS

- avg(expr) − average value for rows within the group
- count(expr) − count of values for rows within the group
- max(expr) − maximum value within the group
- min(expr) − minimum value within the group
- sum(expr) − sum of values within the group

#### EXAMPLE QUERIES

Find out the number of cities:

Find out the number of cities with non-null ratings:

Find out the number of distinctive country values:

SELECT **COUNT**(DISTINCT country_id) FROM city;

Find out the smallest and the greatest country populations:

SELECT **MIN**(population), **MAX**(population)

FROM country;

Find out the total population of cities in respective countries:

SELECT country_id, **SUM**(population)

FROM city GROUP BY country_id;

Find out the average rating for cities in respective countries if the average is above 3.0:

SELECT country_id, **AVG**(rating)

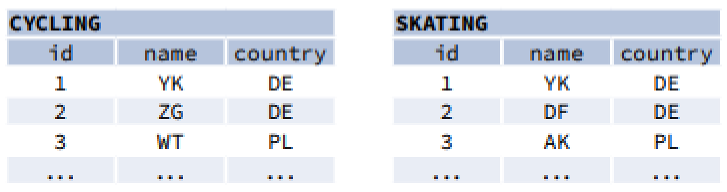
FROM city

GROUP BY country_id

HAVING **AVG**(rating) > 3.0;

## SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.



CYCLING

| id | name | country |
|---|---|---|
| 1 | YK | DE |
| 2 | ZG | DE |
| 3 | WT | PL |
| ... | ... | ... |

SKATING

| id | name | country |
|---|---|---|
| 1 | YK | DE |
| 2 | DF | DE |
| 3 | AK | PL |
| ... | ... | ... |

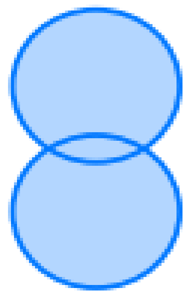
## UNION

UNION combines the results of two result sets and removes duplicates. UNION ALL doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

SELECT name

FROM cycling



WHERE country = 'DE'

**UNION / UNION ALL**

SELECT name

FROM skating

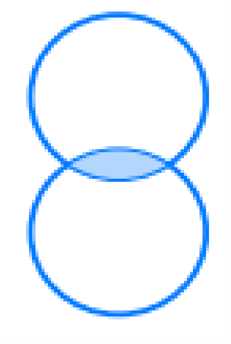
WHERE country = 'DE';

## INTERSECT

INTERSECT returns only rows that appear in both result sets. This query displays German cyclists who are also German skaters at the same time:

SELECT name

FROM cycling



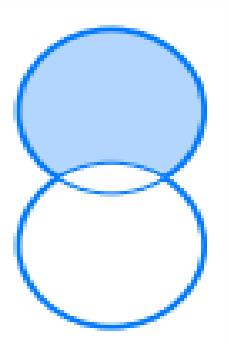
```
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```

## EXCEPT

EXCEPT returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
```



**EXCEPT/MINUS**

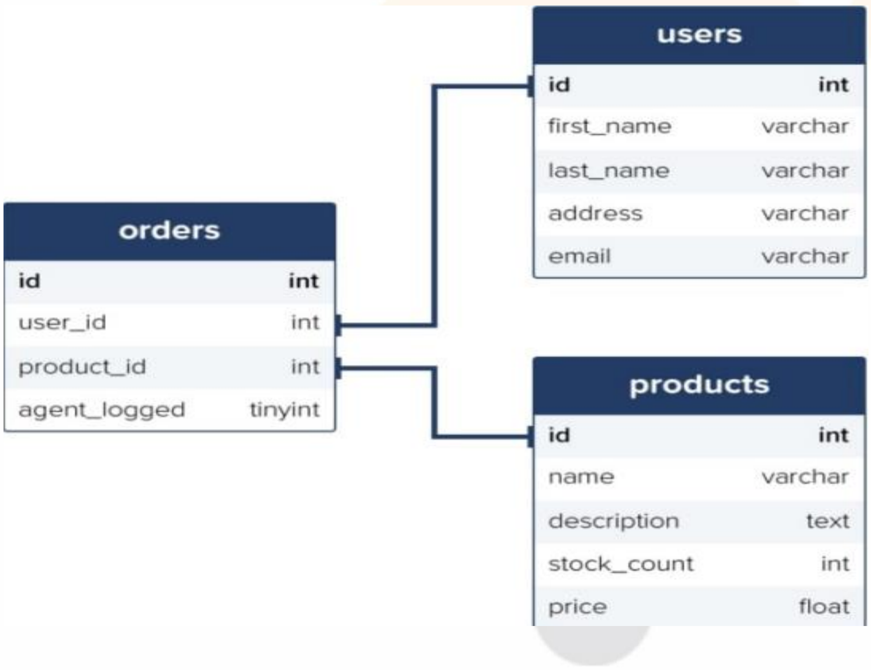
SELECT name

FROM skating

WHERE country = 'DE';

***Let's look at another relational database.***

This example in particular stores e-commerce information, specifically the products on sale, the users who buy them, and records of these orders which link these 2 entities. Suppose the address consists of a city followed by a district.



## WILDCARD CHARACTERS

In SQL, Wildcards are special characters used with the LIKE and NOT LIKE keywords which allow us to search data with sophisticated patterns much more efficiently.

| Wildcards | |
|---|---|
| **Name** | **Description** |
| % | Equates to zero or more characters.<br>**Example 1**: Find all users with surnames ending in 'son'.<br>`SELECT * FROM users`<br>`WHERE surname LIKE '%son';`<br><br>**Example 2**: Find all users living in cities containing the pattern 'che'<br>`SELECT * FROM users`<br>`WHERE city LIKE '%che%';` |
| _ | Equates to any single character.<br>**Example**: Find all users living in cities beginning with any 3 characters, followed by 'chester'.<br>`SELECT * FROM users`<br>`WHERE city LIKE '___chester';` |
| [charlist] | Equates to any single character in the list.<br>**Example 1:** Find all users with first names beginning with J, H or M.<br>`SELECT * FROM users`<br>`WHERE first_name LIKE '[jhm]%';`<br><br>**Example 2:** Find all users with first names beginning letters between A-L.<br>`SELECT * FROM users`<br>`WHERE first_name LIKE '[a-l]%';`<br><br>**Example 3:** Find all users with first names not ending with letters between n-s.<br>`SELECT * FROM users`<br>`WHERE first_name LIKE '%[!n-s]';` |

## KEYS

In relational databases, there is a concept of primary and foreign keys. In SQL tables, these are included as constraints, where a table can have a primary key, a foreign key, or both.

### Primary Key

A primary key allows each record in a table to be uniquely identified. There can only be one primary key per table, and you can assign this constraint to any single or combination of columns. However, this means each value within this column(s) must be unique.

### Foreign Key

A foreign key can be applied to one column or many and is used to link 2 tables together in a relational database. A foreign key also prevents invalid data from being inserted which isn't also present in the parent table.

## MANAGING TABLES

```
CREATE TABLE t
( id INT PRIMARY KEY,
name VARCHAR NOT NULL,
price INT DEFAULT 0 );
```

Create a new table with three columns

```
DROP TABLE t ;
```

Delete the table from the database

```
ALTER TABLE t ADD column;
```

Add a new column to the table

ALTER TABLE t DROP COLUMN c ;

Drop column c from the table

ALTER TABLE t ADD constraint;

Add a constraint

ALTER TABLE t DROP constraint;

Drop a constraint

ALTER TABLE t1 RENAME TO t2;

Rename a table from t1 to t2

ALTER TABLE t1 RENAME c1 TO c2 ;

Rename column c1 to c2

TRUNCATE TABLE t;

Remove all data in a table

## USING SQL CONSTRAINTS

CREATE TABLE t (

c1 INT, c2 INT, c3 VARCHAR,

PRIMARY KEY (c1,c2)

);

Set c1 and c2 as a primary key

```
CREATE TABLE t1(
c1 INT PRIMARY KEY,
 c2 INT,
FOREIGN KEY (c2) REFERENCES t2(c2)
);
```

Set c2 column as a foreign key

```
CREATE TABLE t (
c1 INT, c1 INT, UNIQUE(c2,c3)
);
```

Make the values in c1 and c2 unique

```
CREATE TABLE t(
c1 INT, c2 INT,
CHECK(c1> 0 AND c1 >= c2)
);
```

Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
c1 INT PRIMARY KEY,
c2 VARCHAR NOT NULL
);
```

Set values in c2 column not NULL

## MODIFYING DATA

```
INSERT INTO t(column_list)
 VALUES(value_list);
```

Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
               (value_list), ….;
```

Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```

Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```

Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
       c2 = new_value
WHERE condition;
```

Update values in the column c1, c2 that match the condition

DELETE FROM t;

Delete all data in a table

DELETE FROM t

WHERE condition;

Delete subset of rows in a table

## VIEW

A view is essentially a SQL result set that gets stored in the database under a label, so you can return to it later, without having to rerun the query. These are especially useful when you have a costly SQL query that may be needed a number of times, so instead of running it over and over to generate the same results set, you can just do it once and save it as a view

### Creating Views

To create a view, you can do so like this:

CREATE VIEW **v(c1,c2)**

AS

SELECT c1, c2

FROM t;

Create a new view that consists of c1 and c2

CREATE VIEW **v(c1,c2)**

AS

SELECT c1, c2

FROM t; WITH [CASCADED | LOCAL] CHECK OPTION;

Create a new view with check option

CREATE RECURSIVE VIEW **v**

AS select-statement -- anchor part

UNION [ALL]

select-statement; -- recursive part

Create a recursive view

CREATE TEMPORARY VIEW **v**

AS

 SELECT c1, c2

FROM t;

Create a temporary view

```
CREATE VIEW priority_users AS
SELECT * FROM users
WHERE country = 'United Kingdom';
```

## Replacing Views

With the CREATE OR REPLACE command, a view can be updated.

```
CREATE OR REPLACE VIEW [priority_users] AS

SELECT * FROM users

WHERE country = 'United Kingdom' OR country='USA';
```

## Deleting Views

To delete a view, simply use the DROP VIEW command.

DROP VIEW **view_name**;

Delete a view

```
DROP VIEW priority_users;
```

## INDEXES

Indexes are attributes that can be assigned to columns that are frequently searched against to make data retrieval a quicker and more efficient process.

CREATE INDEX **idx_name**

ON t(c1,c2);

Create an index on c1 and c2 of the table t

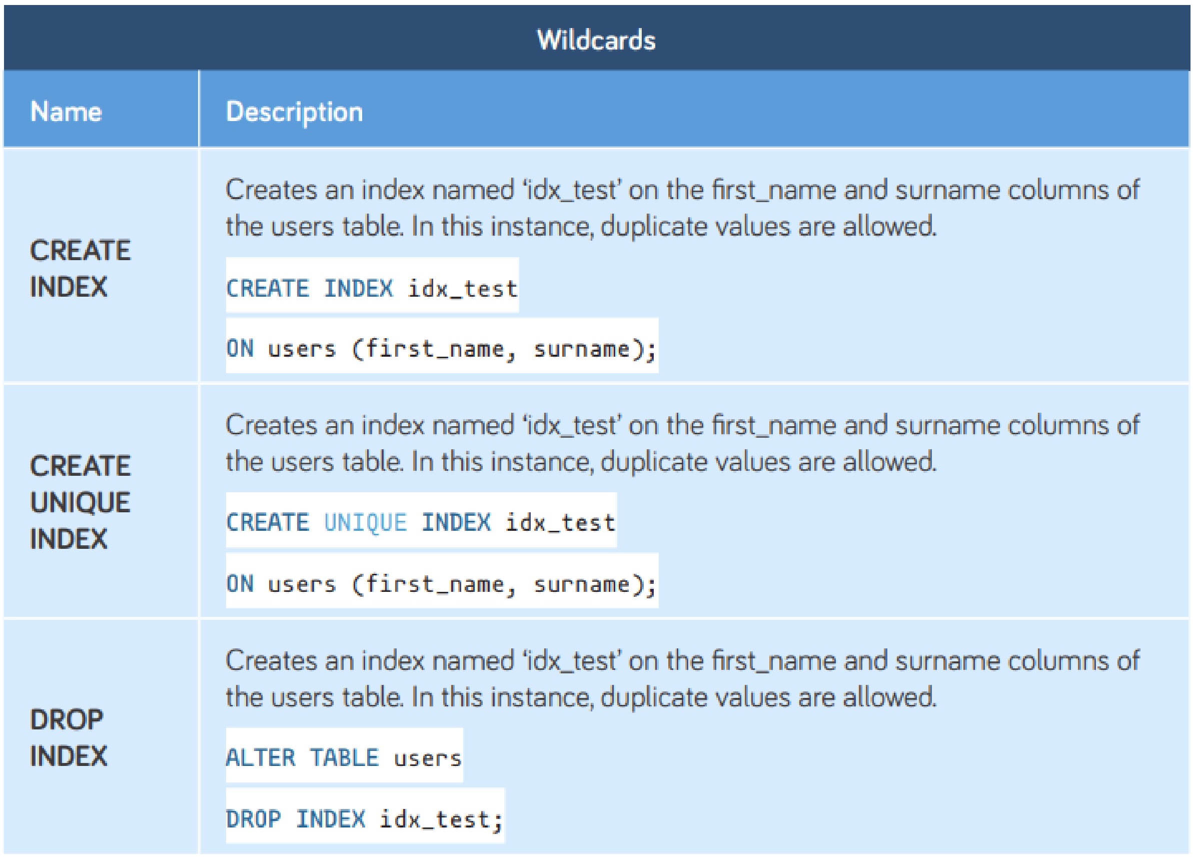
CREATE UNIQUE INDEX **idx_name**

ON t(c3,c4);

Create a unique index on c3, c4 of the table t

DROP INDEX **idx_name**;

Drop an index



| Wildcards | |
|---|---|
| **Name** | **Description** |
| CREATE INDEX | Creates an index named 'idx_test' on the first_name and surname columns of the users table. In this instance, duplicate values are allowed.<br>`CREATE INDEX idx_test`<br>`ON users (first_name, surname);` |
| CREATE UNIQUE INDEX | Creates an index named 'idx_test' on the first_name and surname columns of the users table. In this instance, duplicate values are allowed.<br>`CREATE UNIQUE INDEX idx_test`<br>`ON users (first_name, surname);` |
| DROP INDEX | Creates an index named 'idx_test' on the first_name and surname columns of the users table. In this instance, duplicate values are allowed.<br>`ALTER TABLE users`<br>`DROP INDEX idx_test;` |

## TRIGGERS

A trigger is a piece of code executed automatically in response to a specific event that occurred on a table in the database.

A trigger is always associated with a particular table. If the table is deleted, all the associated triggers are also deleted automatically.

CREATE OR MODIFY TRIGGER **trigger_name**

WHEN EVENT

ON **table_name** TRIGGER_TYPE

EXECUTE **stored_procedure**;

Create or modify a trigger

WHEN-

- BEFORE – invoke before the event occurs
- AFTER – invoke after the event occurs

EVENT-

- INSERT – invoke for INSERT
- UPDATE – invoke for UPDATE
- DELETE – invoke for DELETE

TRIGGER_TYPE

- FOR EACH ROW
- FOR EACH STATEMENT

CREATE TRIGGER **Before_insert_person**

BEFORE INSERT

ON **person** FOR EACH ROW

EXECUTE **stored_procedure**;

Create a trigger invoked before a new row is inserted into the person table.