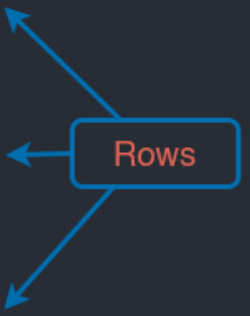
Week 2: Relational Model and Basic SQL

L2.1: *Introduction to Relational Model - Part 1*

Attributes

Example: *Instructor Table*

ID	name	dept_name	salary
1	John Doe	Computer Science	100000
2	Jane Doe	Mathematics	80000
3	Peter Smith	Physics	90000
4	Susan Jones	Chemistry	70000
5	David Brown	Biology	60000
6	Emily Green	English	50000
7	Michael White	History	40000
8	Sarah Black	Art	30000
9	Kevin Blue	Music	20000
10	Ashley Pink	Foreign Languages	10000



- **Attributes** are the column names / fields of a table.
- These values are (normally) required to be atomic (indivisible).
- The set of allowed values for each attribute is called the **domain** of the attribute.
  - **Roll #**: Alphanumeric String
  - **First Name, Last Name**: Alpha String
  - **DoB**: Date
  - **Passport #**: String - nullable (optional)
  - **Aadhaar #**: 12-digit number
  - **Department**: Alpha String
- The special value **null** is a member of every domain, indicates the value is *unknown*.
- The null value may cause complications in the definition of some operations.

Schema and Instance

- $A_1, A_2, ..., A_n$ are attributes.
- $R = (A_1, A_2, ..., A_n)$ is a relation schema.
- Example:
  - *instructor = (ID, name, dept_name, salary)*
- Formally, given sets $D_1, D_2, ..., D_n$ is a **relation** r is a subset of

$$r \in R \subseteq D_1 \times D_2 \times ... \times D_n$$

- Thus a relation is a set of n-tuples $(a_1, a_2, ..., a_n)$ where each $a_i \in D_i$
- An element t of r is a tuple, represented by a row in a table.
- Example:

$$\text{instructor} \equiv (String(5) \times String \times String \times Number+)$$

where:

$$\text{ID } \in String(5),\ \text{name } \in String,\ \text{dept\_name } \in String \text{ and salary } \in Number+$$

- $String(5)$ represents a string of length 5.
- $Number+$ represents a positive number.

Relations are Unordered with Unique Tuples

- Order of tuples / rows is irrelevant.
- No two tuples / row may be identical.

Keys

- Let $K \subseteq R$, where R is the set of attributes in the relation.
- K is a **superkey** of R if K uniquely identifies tuples in R.
- Superkey K is a **candidate key** if K is minimal.
- One of the candidate keys is selected to be the **primary key**.
- A **surrogate key** (synthetic key) in a database is a unique identifier for either an entity in the modeled world or an object in the database.
- $$\text{Students } = \text{Roll\#, First Name, Last Name, DoB, Passport\#, Aadhaar\#, Department}$$

Super Key

- A superkey is a set of attributes that can uniquely identify a row in a table.
- A superkey can contain duplicate values.
- It is not necessarily minimal.
- Example: $\{\text{Roll\#, DoB}\}$
- $Passport\#$ is not because it contains null values.

Candidate Key

- A candidate key is a superkey that is minimal.
- A table can have multiple candidate keys.
- One of the candidate keys is chosen to be the primary key.
- Example: $\{\text{Roll\#, \{First Name, Last Name\}, Aadhaar\#}\}$

Primary Key

- A primary key is a candidate key that is chosen to be unique identifier for a table.
- The primary key must be unique and cannot contain null values.
- The primary key is used to reference rows in other tables.
- Example: $Roll\#$

Surrogate Key

- A surrogate key is an artificial key that is not based on any real-world attribute.
- Surrogate keys has no inherent meaning or relation to the data.
- Surrogate keys are typically integers that are automatically generated by the database.
- Example:
  - *You are a new employee in an IT MNC, your group-lead assigned you to work on a database table. You observed that some entries in the table are duplicate. You can't delete any record but at the same time you must uniquely identify each record so you decided to add a new column in the table which will work as an auto incrementing serial number.*

Secondary Key (Alternate Key)

- A secondary key is a candidate key that is not chosen to be the primary key.
- Secondary keys can be used to create indexes, which can improve the performance of queries.
- Secondary keys cannot be null.
- Example: $\{\text{First Name}, \text{Last Name}\}$, $\text{Aadhaar}\#$

Simple Key

- A simple key is a key that is made up of a single attribute.
- Simple keys are often used when the attriWute is a natural identifier, such as $Roll\#$ or $Aadhaar\#$.

Composite Key

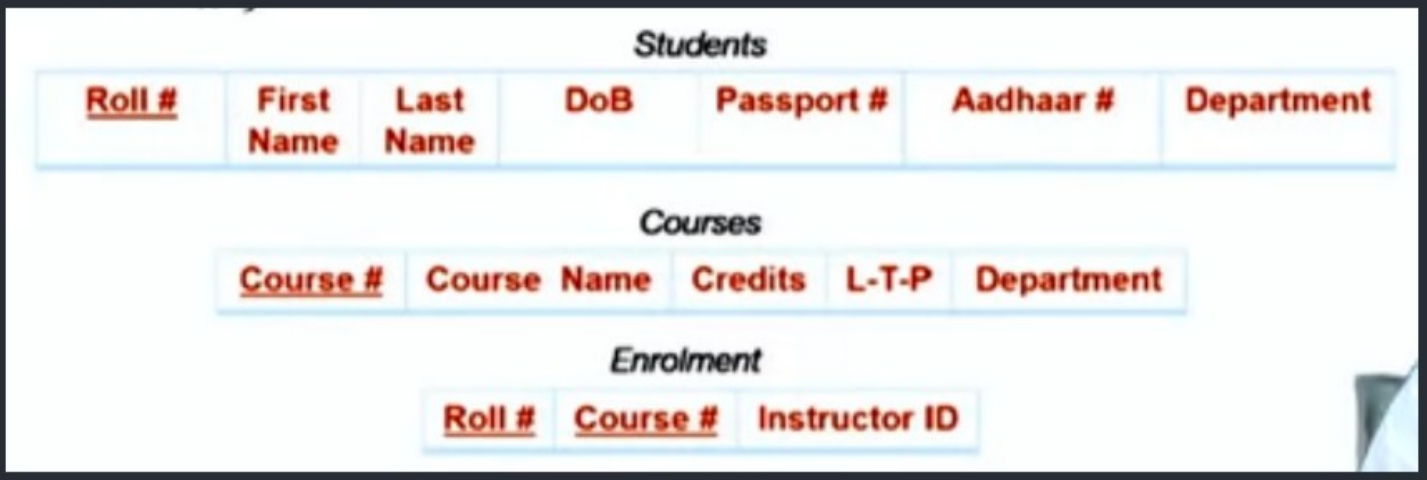
- A composite key is a key that is made up of multiple attributes.
- Composite keys are often used when no single attribute is a unique identifier.
- Composite keys are more flexible than simple keys, but they are be more difficult to understand and manage.
- Example: $\{\text{First Name}, \text{Last Name}\}$

Foreign Key

- A foreign key is a field in one table that references the primary key of another table.
- Foreign keys are used to maintain referential integrity between tables.
- Example:
  - **Referencing** relation:
    - Enrolment: Foreign Keys - $\text{Roll}\#$, $\text{Course}\#$
  - **Referenced** relation:
    - Students, Courses

Compound Key

- A compound key is a key that is made up of multiple attributes.
- Compound keys are often used when no single attribute is a unique identifier.
- Example: $\text{Roll\#}$, $\text{Course\#}$



Additional points

- A **primary key** is always a **super key**.
- A **candidate key** is always a **super key**.
- A **compound key** can be applied to a **foreign key**.
- A **foreign key** can be a simple key or a **candidate key** or **super key**.
- A **foreign key** must always reference the **primary key** of another table.
- A **foreign key** cannot reference itself.

L2.2: *Introduction to Relational Model - Part 2*

Relational Operators

Select Operation - selection of rows (tuples)

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

- $\sigma A = B \wedge D > 5(r)$

A	B	C	D
α	α	1	7
β	β	23	10

Project Operation - selection of columns (attributes)

- Relation r

A	B	C
α	10	1
α	20	1
β	30	1
β	40	2

- $\pi A, C(r)$

A	C
α	1
α	1
β	1
β	2

Union Operation - union of two relations

- Relation r, s

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r \cup s$

A	B
α	1
α	2
β	1
β	3

Difference Operation - difference of two relations

- Relation r, s

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r - s$

A	B
α	1
β	1

$s - r$

A	B
β	3

Intersection Operation - intersection of two relations

- Relation r, s

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

$r \cap s$

A	B
α	2

Cartesian Product Operation - cartesian product of two relations

- Relation r, s

r

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

$r \times s$

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

Renaming a Table

- Allow us to refer to a relation, (say E) by more than one name.

$$\rho_X(E)$$

- Now, E can be referred to by either E or X.

Joining two relations - Natural Join

Let r and s be relations on schemas R and S respectively. The natural join of r and s is on schema $R \cup S$ obtained as follows:

- Consider each pair of tuples t_r, from r and t_s from s.
- If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where:
  - t has the same value as t_r on r.
  - t has the same value as t_s on s.
- Relations r, s

r

A	B	C	D
x	1	2	3
y	4	5	6
z	7	8	9
a	1	3	5

s

B	D	E
1	3	b
4	5	c
1	3	a
1	2	a
1	5	d
1	5	b

Natural Join - $r \bowtie s$

$$\pi A, r.B, C, r.D, E(\sigma r.B = s.B \wedge r.D = s.D(r \times s))$$

A	B	C	D	E
x	1	2	3	b
x	1	2	3	a
a	1	3	5	a
a	1	3	5	b

Aggregate Operations

- SUM
- AVG
- MAX
- MIN

relation r

A	B
a	2
b	6
c	3
d	1

$SUM(\sigma_{B>1}(r))$

- SUM of B where $B > 1$
- Results in 11

L2.3: *Introduction to SQL - Part 1*

History of Query Language

- IBM developed *Structured English Query Language (SEQUEL)* in early 1970s.
- Then renamed it to *Structured Query Language (SQL)*.
- Then in 1986 it is formalized as a standard by ANSI and ISO.

Data Definition Language (DDL)

- Specification notation for defining the database schema
- DDL compiler generates a set of table templates stored in a *data dictionary*.
- Data dictionary contains metadata
  - Database schema
  - Integrity constraints
    - Primary Key (ID uniquely identifies instructors)
  - Authorization
    - Who can access what

Domain Types in SQL (DataTypes)

- `char(n)` - Fixed length character string of length `n`.
- `varchar(n)` - Variable length character string of maximum length `n`.
- `int` - Integer (32 bits)
- `smallint(n)` - Smaller integer with maximum `n` digits.
- `numeric(p, d)` - Fixed point number with `p` digits, `d` of which are after the decimal point.
  - Example: `numeric(5, 2)` can store `-999.99` to `999.99`.
- `float(n)` - Floating point number with `n` digits of precision.

Create Table Construct

1. Start with the `CREATE TABLE` keyword.
2. Specify the name of the table.
3. List the columsn in the table, along with their data types.
4. Optionally, specify constraints for the columns.
5. End the statement with a semicolon.

Example

```sql
CREATE TABLE customers (
  customer_id INT NOT NULL AUTO_INCREMENT,
  first_name VARCHAR(20),
  last_name VARCHAR(10),
  email VARCHAR(50),
  PRIMARY KEY (customer_id)
);
```

This statement creates a table named `customers` with the following columns:

- `customer_id` is an integer that is the primary key of the table.
- `first_name` and `last_name` are strings that store the customer's first and last name.
- `email` is a string that stores the customer's email address.

The `NOT NULL` constraint on the `customer_id` column ensures that this column cannot be null. The `PRIMARY KEY` constraint on the `customer_id` column ensures that this column is unique.

```
CREATE TABLE orders (
  order_id INT AUTO_INCREMENT,
  consumer_id INT NOT NULL,
  order_date DATETIME,
  total_price DECIMAL(10,2),
  PRIMARY KEY (order_id),
  FOREIGN KEY (consumer_id) REFERENCES customers(customer_id)
);
```

This statement creates a table called `orders` with the following columns:

- `order_id` is an integer that is the primary key of the table.
- `consumer_id` is an integer that references the `customer_id` column in the `customers` table.
- `order_date` is a date and time that stores the date and time of the order.
- `total_price` is a decimal number that stores the total price of the order.

`PRIMARY KEY` declaration on an attribute automatically ensures `NOT NULL`.

Update Tables

ALTER

Add a column

```
ALTER TABLE customers
ADD COLUMN phone VARCHAR(15);
```

Modify a column

```
ALTER TABLE customers
MODIFY COLUMN phone VARCHAR(15) NOT NULL;
```

Drop a column

```
ALTER TABLE customers
DROP COLUMN phone;
```

Delete

```
DELETE FROM customers;
```

- Deletes all rows from the table.

Drop

```
DROP TABLE customers;
```

- Deletes the table.

Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model

Insert

```
INSERT INTO customers (first_name, last_name, email)
VALUES ('John', 'Doe', 'johndoe@123.gmail.com`);
```

Update

```
UPDATE customers SET email = 'john@doe.gmail.com'
WHERE customer_id = 1;
```

Delete

```
DELETE FROM customers
WHERE customer_id = 1;
```

Select

- `SELECT` all columns from the table.

  ```
  SELECT * FROM customers;
  ```

- `SELECT` specific columns from the table.

  ```
  SELECT first_name, last_name FROM customers;
  ```

- `SELECT` with a condition.

  ```
  SELECT * FROM customers
  WHERE customer_id = 1;
  ```

- `SELECT` with `DISTINCT`.

  ```
  SELECT DISTINCT first_name FROM customers;
  ```

  - select unique values of `first_name` from the table.

- `SELECT` using `AS`.

  ```
  SELECT first_name AS fname FROM customers;
  ```

    - `AS` helps to create an alias for the column name.
- `SELECT` an attribute with a literal.

```
SELECT 'A' from customers;
```

    - This will select `A` for every row in the table and result in a table with a single column.
- `SELECT` clause can contain arithmetic operations too.

```
SELECT order_id, total_price * 0.18 as tax FROM orders;
```

    - This will select `order_id` and `total_price * 0.18` as `tax` for every row in the table.

WHERE clause

- The where clause specifies conditions that the result must satisfy

```
SELECT * FROM customers
WHERE first_name = 'John' AND last_name = 'Kane';
```

- This will select all the rows from the table where `first_name` is `John` and `last_name` is `Kane`.

FROM clause

- The from clause lists teh relations involved in the query

```
SELECT * FROM customers, orders;
```

- This will select all the rows from the cartesian product of `customers` and `orders`.

L2.4: *Introduction to SQL - Part 2*

Cartesian Product

- The cartesian product of two relations is the set of all tuples that are in the first relation concatenated with all tuples that are in the second relation.

```
SELECT * FROM customers, orders;
```

- This will select all the rows from the cartesian product of `customers` and `orders`.
- i.e. all the rows from `customers` will be concatenated with all the rows from `orders`.
- If there are m rows in `customers` and n rows in `orders`, then the cartesian product will have $m \times n$ rows.

Rename `AS` clause

- The `AS` clause can be used to rename the columns in the result of a query.

```
SELECT first_name AS fname, last_name AS lname FROM customers;
```

- This will select `first_name` as `fname` and `last_name` as `lname` from the `customers` table.

String Operations

- SQL includes a string-matching operator for comparisons on character strings.

`LIKE`

- The `LIKE` operator is used to match a text pattern against a column.
- We can use the `%` wildcard to match any sequence of characters.
- Or, we can use the `_` wildcard to match any single character.

```
SELECT * FROM customers
WHERE first_name LIKE 'J%';
```

- This will select all the rows from the `customers` table where `first_name` starts with `J`.

```
SELECT * FROM customers
WHERE first_name LIKE 'P_r%m';
```

- This will select all the rows from the `customers` table where `first_name` starts with `P`, followed by any single character, followed by `r`, followed by any sequence of characters.
- To match strings having `_` or `%`, we can use the `ESCAPE` clause.

```
SELECT * FROM STUDENTS
WHERE PERCENT LIKE '58!%' ESCAPE '!';
```

- This will select all the rows from the `students` table where `percent` starts with `58%`.
- Any character in `ESCAPE` clause treats as an escape character.

Additional

- SQL supports a variety of string operations:
  - concatenation using `||`

    ```
    SELECT first_name || ' ' || last_name AS name FROM customers;
    ```

    - This will select `first_name` followed by a space followed by `last_name` as `name` from the `customers` table.
  - converting from upper to lower case (and vice-versa).
  - finding string length, extracting substrings, and trimming white space etc...

Ordering

- The `ORDER BY` clause is used to sort the result in ascending or descending order.
- By default, the `ORDER BY` clause sorts in ascending order.

```
SELECT * FROM customers
ORDER BY first_name DESC;
```

- This will select all the rows from the `customers` table and sort them in descending order of `first_name`.

Selecting number of rows in output

- To select the first 5 rows from the `customers` table.

MYSQL

```sql
SELECT * FROM customers
LIMIT 5;
```

SQL Server & MS Access

```sql
SELECT TOP 5 * FROM customers;
```

Oracle

```sql
SELECT * FROM customers
FETCH FIRST 5 ROWS ONLY;
```

PostgreSQL

```sql
SELECT * FROM customers
LIMIT 5 OFFSET 0;
```

OFFSET	Meaning
0	Skip 0 rows.
1	Skip 1 row.
10	Skip 10 rows.
−1	Skip the last row.
−10	Skip the last 10 rows.

Where Clause Predicates

`BETWEEN`

- The `BETWEEN` operator is used to match a value against a range of values.

```sql
SELECT * FROM orders
WHERE total_price BETWEEN 1000 AND 2000;
```

- This will select all the rows from the `orders` table where `total_price` is between 1000 and 2000.
- It will include 1000 and 2000 in the result, i.e. ($\geq$ 1000 and $\leq$ 2000).

`IN`

- The `IN` operator is used to match a value against a set of values.

```
SELECT * FROM orders
WHERE total_price IN (1000, 2000, 3000);
```

- This will select all the rows from the `orders` table where `total_price` is either 1000, 2000 or 3000.

Tuple Comparison

- We can compare tuples using the comparison operators.

```
SELECT * FROM orders
WHERE (order_date, total_price) > ('2019-01-01', 1000);
```

- This will select all the rows from the `orders` table where `order_date` is greater than $2019-01-01$ and `total_price` is greater than 1000.

> And there are many more...

L2.5: *Introduction to SQL - Part 3*

Set Operations

Union

- The union of two relations is the set of all tuples that are in either relation.

```
(SELECT order_id FROM orders
WHERE YEAR(order_date) = 2019
AND total_price > 1000)

UNION

(SELECT order_id FROM orders
WHERE YEAR(order_date) = 2020
AND total_price > 2000);
```

- This will select all the `order_id` from the `orders` table where `order_date` is in 2019 and `total_price` is greater than 1000 and `order_id` from the `orders` table where `order_date` is in 2020 and `total_price` is greater than 2000.

Intersect

- The intersect of two relations is the set of all tuples that are in both relations.

```
(SELECT consumer_id FROM orders
WHERE YEAR(order_date) = 2019)

INTERSECT

(SELECT consumer_id FROM orders
WHERE YEAR(order_date) = 2020);
```

- This query will select all the `consumer_id` from the `orders` table where `order_date` is in 2019 and `consumer_id` from the `orders` table where `order_date` is in 2020.
- Basically `consumer_id` of those consumers who ordered something in both 2019 and 2020.

Except

- The except of two relations is the set of all tuples that are in the first relation but not in the second relation.

```
(SELECT consumer_id FROM orders
WHERE YEAR(order_date) > 2019)

EXCEPT

(SELECT consumer_id FROM orders
WHERE YEAR(order_date) = 2022
AND total_price < 1000);
```

> Suppose a tuple occurs `m` times in `r` and `n` times in `s`, then it occurs:
>
> - $m + n$ times in $r \cup s$ (`r UNION s`)
> - $min(m, n)$ times in $r \cap s$ (`r INTERSECT s`)
> - $maxn(0, m - n)$ times in $r - s$ (`r EXCEPT s`)

NULL VALUES

Meaning Nulls

- Null represents the lack of a value or unknown information in a column for a particular row.
- It is not the same as an empty string or zero.

Handling Nulls

- Nulls can be assigned or exist naturally in a column. They can be handled by using special operators like `IS NULL` and `IS NOT NULL` to check for the presence or absence of null values.
- Comparisons involving null values using equality (`=`) or inequality (`<>` , `!=`) operators typically result in an unknown or null result.

Nullable Columns

- Columns can be specified as nullable or non-nullable during table creation.
- A nullable column allows null values, while a non-nullable column requires a valid value for each row.

Effects on Operations

- Null values have specific behaviors in SQL operations.
- Arithmetic calculations involving null will typically result in null.
- Concatenating a null value with a non-null value will yield a null result.
- Aggregate functions, such as `SUM`, `COUNT`, `AVG`, ignore null values by default.

Handling Nulls in Quries

- SQL provides functions like `COALESCE` and `ISNULL` to handle null values in queries.

`COALESCE`

```
SELECT COALESCE(last_name, 'N/A') as lname
FROM customers;
```

- This will select `last_name` from the `customers` table and if `last_name` is null then it will select `N/A` instead.

`ISNULL`

```
SELECT ISNULL(last_name, 'N/A') AS lname
FROM customers;
```

- This will select `last_name` from the `customers` table and if `last_name` is null then it will select `N/A` instead.

Null values: Three Valued Logic

- Three values: **true, false, unknown**
- Any comparison with null returns known

```
5 < null or nul <> null or null = null
```

	`true`	`false`	`NULL`
`true`	`OR = true` `AND = true`	`OR = true` `AND = false`	`OR = true` `AND = NULL`
`false`	`OR = true` `AND = false`	`OR = false` `AND = false`	`OR = NULL` `AND = false`
`NULL`	`OR = true` `AND = true`	`OR = true` `AND = true`	`OR = NULL` `AND = NULL`

- `(NOT NULL) = NULL`

Aggregate Functions

- Aggregate functions are used to compute a single result from a set of input values.
  - `COUNT` - returns the number of rows in the input
  - `SUM` - returns the sum of all values in the input
  - `AVG` - returns the average of all values in the input

    - `MIN` - returns the minimum value in the input
    - `MAX` - returns the maximum value in the input
- Aggregate functions are often used with the `GROUP BY` clause of the `SELECT` statement.

GROUP BY

- The `GROUP BY` clause is used to group rows with matching values in one or more columns.

```
SELECT consumer_id, COUNT(*) AS num_orders
FROM orders
GROUP BY consumer_id;
```

- This will return the number of orders made by each consumer.

```
SELECT consumer_id, YEAR(order_date) as order_year FROM orders
WHERE total_price = MAX(total_price);
GROUP BY YEAR(order_date);
```

- This will return the `consumer_id` of the consumer who made the maximum order in each year.

HAVING

- The `HAVING` clause is used to filter groups of rows.

```
SELECT consumer_id, COUNT(*) AS num_orders
FROM orders
GROUP BY consumer_id
HAVING COUNT(*) > 1;
```

- This will return the `consumer_id` of the consumers who made more than one order.

```
SELECT consumer_id, COUNT(*) AS num_orders
FROM orders
WHERE YEAR(order_date) > 2020
GROUP BY consumer_id
HAVING COUNT(*) > 5;
```

- This will return the `consumer_id` of the consumers who made more than five orders from 2021.

ORDER OF QUERIES IN SQL

`SELECT` > `FROM` > `[JOIN]` > `WHERE` > `GROUP BY` > `HAVING` > `ORDER BY`

Additional Resources (Cheatsheet)

PostgreSQL 🔗

MySQL 🔗

SQL Basics Cheat Sheet

LearnSQL.com

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;
```

Fetch `id` and `name` columns from the `city` table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the `rating` column in the default ASCending order:

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

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

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

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

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, `NULL`s 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, `NULL`s 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, `NULL`s 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` used these columns to match rows: **`city.id, city.name, country.id, country.name`**
`NATURAL JOIN` is very rarely used in practice.

SQL Basics Cheat Sheet

LearnSQL.com

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:

```
SELECT COUNT(*)
FROM city;
```

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

```
SELECT COUNT(rating)
FROM city;
```

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;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.

This query finds cities with the same rating as Paris:

```
SELECT name FROM city
WHERE rating = (
    SELECT rating
    FROM city
    WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators IN, EXISTS, ALL, or ANY.

This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
    SELECT country_id
    FROM country
    WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
    SELECT AVG(population)
    FROM city average_city
    WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
    SELECT *
    FROM city
    WHERE country_id = country.id
);
```

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

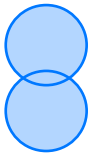
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

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';
```

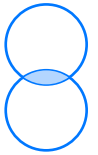


INTERSECT

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';
```