Data & Applications Tutorial - 5

Akshit Sinha
Ayush Agrawal

BASICS OF SQL

- SQL - Structured Query Language
- It is a standard database language that is used to create, maintain, and retrieve the data from relational databases. Relational databases are based on the relational model, an intuitive, straightforward way of representing data in tables. Eg. MySQL , Postgre etc.
- Some points about about SQL:
  - Most common language for extracting and organising data that is stored in a relational database
  - Case insensitive. Eg. select * from TABLE1; select * frOM TABLE1; (Note that entity names like the table name are case sensitive.). It is still recommended to stick to either all caps or all lowercase for sql syntax to avoid confusion.
  - SQL has a universal standard , although exact implementation across different systems might vary.

DATABASE LEVEL COMMANDS

Creating a database

```
CREATE DATABASE <DATABASE NAME>;
```

To list all databases created in a tabular form

```
SHOW DATABASES;
```

Using/Accessing a database to work on

```
USE <DATABASE NAME>;
```

Deleting a database

```
DROP DATABASE <DATABASE NAME>
```

TABLE LEVEL COMMANDS

- A table is what represents a **Relation**.
- Each row in a table is called a **Tuple**.
- Number of columns in a table is called the **Degree**.
- The number of unique rows in a table is the **Cardinality** of the table.
- Creating a table:

```
CREATE TABLE table_name(
column1 datatype [constraints if any],
column2 datatype [constraints if any],
column3 datatype [constraints if any], .....
columnN datatype [constraints if any],
PRIMARY KEY( one or more columns ) ) [other out of line constraints];
```

- The Primary key can either be put at the end (out of line) as above or next to the columns (inline).
- The datatypes can be CHAR , VARCHAR , INT , DECIMAL etc.
- We can use other constraints in the table such as NOT NULL (no null values allowed) , UNIQUE (no duplicates allowed) etc.

TABLE LEVEL COMMANDS

Describing a table: Viewing the table's metadata

`DESC <TABLE NAME>;`
Eg: `DESC CUSTOMERS;`

Field	Type	Null	Key	Default	Extra
ID	int	NO	PRI	NULL	
NAME	varchar(20)	NO		NULL	
AGE	int	NO		NULL	
ADDRESS	char(25)	YES		NULL	
SALARY	decimal(18,2)	YES		NULL	

Deleting a table

`DROP TABLE <TABLE NAME>;`
(is auto committed so cannot be roll backed easily)

TABLE LEVEL COMMANDS: INSERTION

There are two ways of inserting into a table:

1. Inserting values as per order inferred from the order of columns in the table (implicit)

   ```
   INSERT INTO <TABLE NAME> VALUES(v1 , v2 , v3 ,.... vn);
   ```

2. Explicitly specifying the columns and their order (gives more flexibility)

   ```
   INSERT INTO <TABLE NAME>(column1 , column2 ... columnn) VALUES(v1 , v2 , ... vn);
   ```

Note:

To avoid inserting a numerical increasing primary key with each new row, you can set that primary key to auto increment using `AUTO INCREMENT` key word in while creating the table.

Eg: `CREATE TABLE CUSTOMERS (ID INT AUTO INCREMENT PRIMARY KEY , ...)`

TABLE LEVEL COMMANDS: MODIFICATIONS

Add a column to a table

`ALTER TABLE table_name ADD column_name datatype;`

Eg: `ALTER TABLE CUSTOMERS ADD EMAIL varchar(255);`

Drop a column of a table

`ALTER TABLE table_name DROP COLUMN column_name;`

Eg: `ALTER TABLE CUSTOMERS DROP COLUMN EMAIL;`

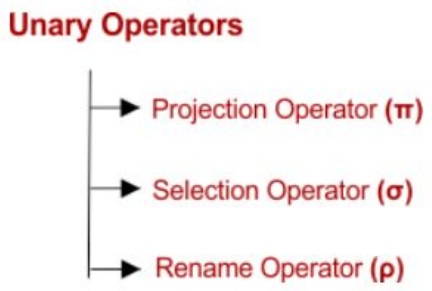
Modify a column of a table

`ALTER TABLE table_name MODIFY COLUMN column_name datatype;`

Eg: `ALTER TABLE CUSTOMERS MODIFY COLUMN EMAIL varchar(511);`

UNARY RELATIONAL OPERATIONS
SELECTION OPERATION

The `SELECT` operation is used to choose a subset of the tuples from a relation that satisfies a selection condition.



In general, the `SELECT` operation is denoted by:

$$\sigma < select\_condition > (R)$$

Eg: $\sigma Salary > 30000(EMPLOYEE)$

We can also have multiple conditions joined by logical operators:

$$\sigma(Dno = 4\ AND\ Salary > 25000) OR (Dno = 5\ AND\ Salary > 30000)(EMPLOYEE)$$

Notice that the `SELECT` operation is commutative; that is,

$$\sigma < cond1 > (\sigma < cond2 > (R)) = \sigma < cond2 > (\sigma < cond1 > (R))$$

SELECTION: EXAMPLE

Query all attributes of every Japanese city in the **CITY** table. The **COUNTRYCODE** for Japan is JPN.

The **CITY** table is described as follows:

CITY

Field	Type
ID	NUMBER
NAME	VARCHAR2(17)
COUNTRYCODE	VARCHAR2(3)
DISTRICT	VARCHAR2(20)
POPULATION	NUMBER

SOLUTION

```
SELECT * FROM CITY WHERE CountryCode = 'JPN';
```

PROJECT OPERATION

The PROJECT operation selects certain columns from the table and discards the other columns.

The general form of the PROJECT operation is

$$\pi < attributelist > (R)$$

The PROJECT operation removes any duplicate tuples, so the result is a set of distinct tuples and hence, a valid relation. This is known as duplicate elimination.

It is also noteworthy that commutativity does not hold on PROJECT.
=> (π <list2> (π <list1> (R)) ≠ π <list1> (π <list2> (R))

The degree (number of attributes) of resulting relation from a project operation is equal to the number of attribute in the attribute list.

PROJECTION: EXAMPLE

Query the **NAME** field for all American cities in the **CITY** table with populations larger than 120000. The *CountryCode* for America is USA.

The **CITY** table is described as follows:

CITY

Field	Type
ID	NUMBER
NAME	VARCHAR2(17)
COUNTRYCODE	VARCHAR2(3)
DISTRICT	VARCHAR2(20)
POPULATION	NUMBER

SOLUTION

```
SELECT NAME FROM CITY WHERE COUNTRYCODE = 'USA' AND
POPULATION > 120000;
```

RENAME OPERATION

In general, for most queries, we need to apply several relational algebra operations one after the other.

We can create intermediate results that store the results of one operation. We must name these intermediate results.

Eg: $\pi Fname, Lname, Salary(\sigma Dno = 5(EMPLOYEE))$

Using Rename Operation: $DEP5\_EMPS \leftarrow \sigma Dno = 5(EMPLOYEE)$

This gives: $RESULT \leftarrow \pi Fname, Lname, Salary(DEP5\_EMPS)$

The general `RENAME` operation when applied to a relation *R* of degree *n* is denoted by any of the following three forms:

$$\rho_{S(B_1,B_2,\ldots,B_n)}(R)$$

or

$$\rho_S(R)$$

or

$$\rho_{(B_1,B_2,\ldots,B_n)}(R)$$

RENAMING: EXAMPLE

Query ***Branch*** using ***Stream*** as alias name and ***Grade*** as CGPA from table ***Student_Details***.

Student_Details

ROLL_NO	Branch	Grade
1	Information Technology	O
2	Computer Science	E
3	Computer Science	O
4	Mechanical Engineering	A

SOLUTION

```
SELECT Branch AS Stream, Grade as CGPA FROM Student_Details;
```

Stream	CGPA
Information Technology	O
Computer Science	E
Computer Science	O
Mechanical Engineering	A

GROUP BY - HAVING Clause

```
SELECT <…> GROUP BY column_name1[,column_name2,...] [HAVING condition];
```

- Used for grouping selections by all values of a column (eg: gender of employee)
- Can use HAVING clause to restrict grouping to only some values of a column (eg: only female employees)
- Commonly used for aggregation purposes

Aggregations

```
AggregateFunction(DISTINCT or ALL GroupName)
```

- Aggregations: COUNT(), MIN(), MAX(), SUM() etc.
- Used with GROUP BY clause to calculate the above outputs for columns you want to group by
- Eg: Number of employees per gender :
    - Query: SELECT COUNT(*) FROM EMPLOYEES GROUP BY GENDER;
- Eg: Number of female employees:
    - Query: SELECT COUNT(*) FROM EMPLOYEES GROUP BY GENDER HAVING GENDER=”F”;

Nested queries

```
SELECT column1, column2, ...

FROM table1

WHERE column1 IN

    ( SELECT column1

      FROM table2

      WHERE condition );
```

Nested queries

```
SELECT column1, column2, ...
FROM table1
WHERE column1 IN
```

Outer Query

```
( SELECT column1
  FROM table2
  WHERE condition );
```

Inner Query

- Inner queries are executed first
- Inner queries return a set of values (in this case set of values of column1)
- The outer query can then use this set of values to check a condition
- *Very helpful* when dealing with multiple tables

Nested queries

Example: Find the names of all employees who have made sales greater than $3000.

Query:

```
SELECT emp_name
FROM employees
WHERE emp_id = ALL (SELECT emp_id
   FROM sales
   WHERE sale_amt > 3000);
```

emp_id	emp_name	dept_id
1	John	1
2	Mary	2
3	Bob	1
4	Alice	3
5	Tom	1

sale_id	emp_id	sale_amt
1	1	1000
2	2	2000
3	3	3000
4	1	4000
5	5	5000
6	3	6000
7	2	7000

OPERATIONS FROM SET THEORY

- A relation is defined as a set of tuples. Hence, we can apply all standard set theory operations on relations.
- Several set theoretic operations are used to merge the elements of two sets in various ways, including UNION, INTERSECTION, and SET DIFFERENCE (also called MINUS or EXCEPT).
- These are binary operations; that is, each is applied to two sets.
- The two relations on which any of these three operations are applied must have the same type of tuples; this condition has been called union compatibility.

OPERATIONS FROM SET THEORY

A. **UNION**

The result of this operation, denoted by $R \cup S$, is a relation that includes all tuples that are either in R or in S or in both R and S. Duplicate tuples are eliminated.

B. **INTERSECTION**

The result of this operation, denoted by $R \cap S$, is a relation that includes all tuples that are in both R and S.

C. **SET DIFFERENCE (or MINUS)**

The result of this operation, denoted by $R - S$, is a relation that includes all tuples that are in R but not in S.

SET OPERATIONS IN SQL

1. Union:
   `SELECT column_name FROM table1 UNION SELECT column_name FROM table2;`
   The union operation eliminates the duplicate rows from its resultset.

2. Union All:
   `SELECT column_name FROM table1 UNION ALL SELECT column_name FROM table2;`
   Union All operation is equal to the Union operation. It returns the set without removing duplication.

3. Intersection:
   `SELECT column_name FROM table1 INTERSECT SELECT column_name FROM table2;`

4. Difference:
   `SELECT column_name FROM table1 MINUS SELECT column_name FROM table2;`

UNION: EXAMPLE

Table: Teachers

id	name	age
1	Peter	32
2	Megan	43
3	Rose	43
4	Linda	30

Table: Students

id	name	age
1	Harry	23
2	Jack	42
3	Joe	32
4	Dent	23

```
SELECT age
FROM Teachers
UNION
SELECT age
FROM Students;
```

age
23
30
32
42
43

CARTESIAN PRODUCT

- CARTESIAN PRODUCT operation—also known as CROSS PRODUCT or CROSS JOIN—is denoted by ×.
- This is also a binary set operation, but the relations on which it is applied do not have to be union compatible.
- Combines every tuple in R to every tuple in S.
- In general, the result of R(A1, A2, ... , A n) × S(B1, B2, ... , B m) is a relation Q with degree n + m attributes Q(A1, A2, ... , An, B1 , B2 , ... , Bm), in that order.
- The CARTESIAN PRODUCT operation applied by itself is generally meaningless. It is mostly useful when followed by a selection that matches values of attributes coming from the component relations.

CARTESIAN PRODUCT IN SQL (CROSS JOIN)

Select ***NAME*** and ***Age*** from ***Student*** table and ***COURSE_ID*** from ***StudentCourse*** table.

Student

ROLL_NO	NAME	ADDRESS	PHONE	Age
1	Ram	Delhi	XXXXXXXXXX	18
2	RAMESH	GURGAON	XXXXXXXXXX	18
3	SUJIT	ROHTAK	XXXXXXXXXX	20
4	SURESH	Delhi	XXXXXXXXXX	18

StudentCourse

COURSE_ID	ROLL_NO
1	1
2	2
2	3
3	4

SOLUTION

```
SELECT Student.NAME, Student.AGE,StudentCourse.COURSE_ID
FROM Student CROSS JOIN StudentCourse;
```

NAME	AGE	COURSE_ID
Ram	18	1
Ram	18	2
Ram	18	2
Ram	18	3
RAMESH	18	1
RAMESH	18	2
RAMESH	18	2
RAMESH	18	3
SUJIT	20	1
SUJIT	20	2
SUJIT	20	2
SUJIT	20	3
SURESH	18	1
SURESH	18	2
SURESH	18	2
SURESH	18	3

JOIN OPERATION

We can select related tuples only from the two relations by specifying an appropriate selection condition after the Cartesian product.

Since this sequence of CARTESIAN PRODUCT followed by SELECT is quite commonly used to combine related tuples from two relations, a special operation, called JOIN, was created to specify this sequence as a single operation.

The JOIN operation, denoted by $\bowtie$, is used to combine related tuples from two relations into single “longer” tuples.

Eg: $\text{DEPT\_MGR} \leftarrow \text{DEPARTMENT}_{\text{Mgr\_ssn=Ssn}}\ \text{EMPLOYEE}$

$\text{RESULT} \leftarrow \pi_{\text{Dname, Lname, Fname}}(\text{DEPT\_MGR})$

The general form of a JOIN operation on two relations $R(A_1, A_2, \ldots, A_n)$ and $S(B_1, B_2, \ldots, B_m)$ is $R_{\langle \text{join condition} \rangle} S$.

JOIN: EXAMPLE

Given the **CITY** and **COUNTRY** tables, query the names of all cities where the *CONTINENT* is *'Africa'*.

COUNTRY

Field	Type
CODE	VARCHAR2(3)
NAME	VARCHAR2(44)
CONTINENT	VARCHAR2(13)
REGION	VARCHAR2(25)
SURFACEAREA	NUMBER
INDEPYEAR	VARCHAR2(5)
POPULATION	NUMBER
LIFEEXPECTANCY	VARCHAR2(4)
GNP	NUMBER
GNPOLD	VARCHAR2(9)
LOCALNAME	VARCHAR2(44)
GOVERNMENTFORM	VARCHAR2(44)
HEADOFSTATE	VARCHAR2(32)
CAPITAL	VARCHAR2(4)
CODE2	VARCHAR2(2)

CITY

Field	Type
ID	NUMBER
NAME	VARCHAR2(17)
COUNTRYCODE	VARCHAR2(3)
DISTRICT	VARCHAR2(20)
POPULATION	NUMBER

SOLUTION

```
SELECT CITY.NAME FROM CITY JOIN COUNTRY ON CITY.COUNTRYCODE
= COUNTRY.CODE WHERE COUNTRY.CONTINENT = 'Africa'
```

```
Qina
Warraq al-Arab
Kempton Park
Alberton
Klerksdorp
Uitenhage
Brakpan
Libreville
```

if-then-else FLOW IN SQL

Write a query identifying the type of each record in the TRIANGLES table using its three side lengths. Output one of the following statements for each record in the table:

Equilateral, Isosceles, Scalene, or Not a Triangle

Column	Type
A	Integer
B	Integer
C	Integer

A	B	C
20	20	23
20	20	20
20	21	22
13	14	30

Isosceles
Equilateral
Scalene
Not A Triangle

CASE STATEMENT

```
SELECT
CASE
    WHEN A+B<=C OR B+C<=A OR C+A<=B THEN "Not A Triangle"
    WHEN A=B AND A=C THEN "Equilateral"
    WHEN A=B OR A=C OR B=C THEN "Isosceles"
    ELSE "Scalene"
END
FROM TRIANGLES
```

THANK YOU