Introduction to SQL

Introduction

What is a Database

A database is a structured collection of data. It is designed to efficiently store, retrieve, and manage information.

The most widely used type of databases are relational databases (PostgresSQL, MySQL, SQL Server, Oracle, etc.). In a relational database, data is stored in tables (rows and columns) and the tables can be have relationships with other tables. For example, a Patient Demographic table will be related to a Patient Medication table through the Patient ID.

About SQL

'SQL' (Structured Query Language) is the language used to manipulate data is relational databases. SQL can be used to Create, Read, Update, and Delete data within relational databases.

Databases that use SQL

- SQL Server
- Oracle
- Postgres SQL
- MySQL

- DB2
- ... and more

A lot of the big data structures and non-relational databases are also incorporating SQL-like syntax so that users can query their databases.

GUI Tool for SQL

- DBeaver (Paid and Free editions) https://dbeaver.io/download/

Tutorial Prerequisites

Basic understanding of tabular data (like Excel).

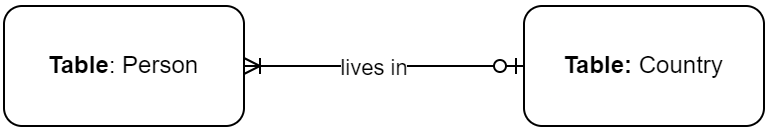
Project Setup

We will use the following site to test SQL commands: https://sqliteonline.com/

Connect to the 'PostgreSQL' database. Note that this site has a 15 minute idle limit.

Commands to insert data

These commands are also explained in later sections of the tutorial. For teaching purposes, they have been replicated for project set up so that users can learn how to query data first (i.e. the most likely step that users are performing).




```
-- Create the following two tables for this demo:
CREATE TABLE country (
        id INTEGER PRIMARY KEY,
        country_name VARCHAR(100)
);

CREATE TABLE person (
        id INTEGER PRIMARY KEY,
        person_name VARCHAR(100),
        income NUMERIC(7,2),
        country_id integer,
        CONSTRAINT country_id_fk
                FOREIGN KEY(country_id)
                REFERENCES country(id)
);

-- Insert data into the 'country' table:
INSERT INTO country (id, country_name) VALUES (1, 'Canada'), (2, 'USA'), (3, 'Mexico'), (4, 'Sweden');

-- Insert data into the 'person' table:
INSERT INTO person (id, person_name, income, country_id) VALUES (1, 'Sally', 60000, 1);
INSERT INTO person (id, person_name, income, country_id) VALUES (2, 'Bob', 70000, 1);
INSERT INTO person (id, person_name, income, country_id) VALUES (3, 'Lucy', 80000, 2);
INSERT INTO person (id, person_name, income, country_id) VALUES (4, 'Bill', 75000, NULL);
```

Result - Person Table

id	person_name	income	country_id
1	Sally	60000.0	1
2	Bob	70000.0	1
3	Lucy	80000.0	2
4	Bill	75000.0	NULL

Result - Country Table

id	country_name
1	Canada
2	USA
3	Mexico
4	Sweden

The two red columns (person.country_id and country.id) is meant to indicate a foreign-key relationship. For example, it can be read as: "Sally lives in country USA". There are different benefits for having the information in separate tables but the main reason is to reduce redundant data.

Basic Data Querying

Select Statement

```
-- select all columns from person table

select
        *
from
        person;
```

id	person_name	income	country_id
1	Sally	60000.0	1
2	Bob	70000.0	1
3	Lucy	80000.0	2
4	Bill	75000.0	NULL

```
-- we can also specify the columns we want to query by replacing the asterisk with the column name

select
        id,
        person_name,
        income
from
        person;
```

id	person_name	income
1	Sally	60000.0
2	Bob	70000.0
3	Lucy	80000.0
4	Bill	75000.0

'Order by' clause

```
-- if we want to return the data and have it ordered according to a specific column(s), we can use the order
by clause

select * from person
order by income asc;

select * from person
order by income desc;
```

id	person_name	income	country_id
1	Sally	60000.0	1
2	Bob	70000.0	1
4	Bill	75000.0	NULL
3	Lucy	80000.0	2

id	person_name	income	country_id
3	Lucy	80000.0	2
4	Bill	75000.0	NULL
2	Bob	70000.0	1
1	Sally	60000.0	1

'Where' Clause

```
-- filter data from person table by name
select * from person where person_name = 'Lucy';

-- sometimes we don't know the exact spelling or word.  In that case we can use a wildcard
select * from person where person_name like '%Lu%';
```

id	person_name	income	country_id
3	Lucy	80000.0	2

'Distinct' Clause

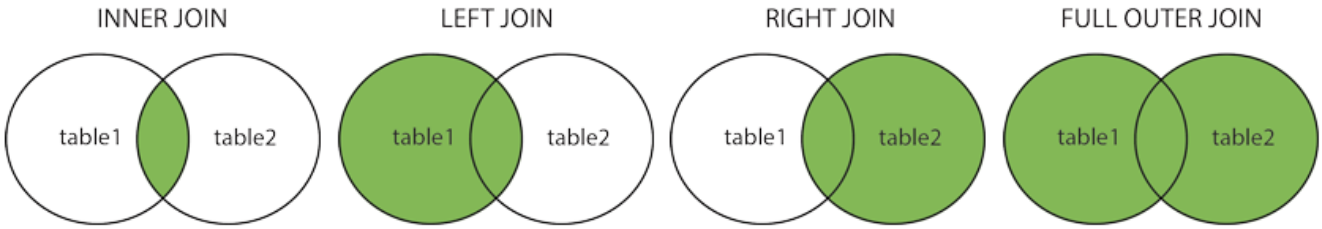
```
-- get the unique countries that a person lives in

select distinct country_id from person;
```

country_id
1
2
NULL

Join Types

There are 3 types of joins that are available in SQL. The following is a good diagram that explains the results from each join type: The diagram and more information can be found here: https://www.w3schools.com/sql/sql_join.asp



Data in relational databases can be spread across many tables in the database. This is done to enable efficiency at different stages of data maintenance process (writing, querying, updates, etc). Join clauses allow us to merge different columns from different tables based on a join condition.

Inner Join

```
select
        *
from
        person
inner join country
        on person.country_id = country.id;
```

Return records that are matching in both person and country table.

id	person_name	income	country_id	id
1	Sally	60000.0	1	1
2	Bob	70000.0	1	1
3	Lucy	80000.0	2	2

Left Join

```
select
        *
from
        person
left join
        country
        on person.country_id = country.id;
```

Returns all records from the person table and only the matching records from the country table.

id	person_name	income	country_id	id
1	Sally	60000.0	1	1
2	Bob	70000.0	1	1
3	Lucy	80000.0	2	2
4	Bill	75000.0	NULL	NULL

Right Join

```
--now do a right join
select *
from
        person
right join
        country
        on person.country_id = country.id;
```

Returns only matching records from the person table and all records from the country table.

id	person_name	income	country_id	id
1	Sally	60000.0	1	1
2	Bob	70000.0	1	1
3	Lucy	80000.0	2	2
				4
				3

Full Outer Join

```
select *
from
        person
full outer join
        country
        on person.country_id = country.id;
```

Returns all records from person and country table.

id	person_name	income	country_id	id
1	Sally	60000.0	1	1
2	Bob	70000.0	1	1
3	Lucy	80000.0	2	2
4	Bill	75000.0	NULL	
				4
				3

Group By and Aggregation

Select with Group By clause

The 'GROUP BY' clause in SQL is used when you want to group rows that have the same values in certain columns and perform some kind of aggregate function (avg, max, min, count) on each group. This is particularly useful when analyzing and summarizing data. Common usage scenarios include:

- calculating summaries
- data categorization
- reporting and visualization

For example, here we are asking the database to return the count of people per country.

```
select
        country_name,
        count(*)
from
        person
inner join
        country
        on person.country_id = country.id
group by
        country_name;
```

country_name	count
USA	1
Canada	2

Having Clause

Using the 'having' clause, we can take the result of the group by statement and filter it to only return countries where the number of people is greater than 1.

```
select
        country_name,
        count(*)
from
        person
inner join
        country
                on person.country_id = country.id
group by
        country_name
having
        count(*) > 1;
```

country_name	count
Canada	2

Average

Return the average income per person per country.

```
select
        country_name,
        avg(income)
from
        person
inner join
        country
                on person.country_id = country.id
group by
        country_name;
```

country_name	avg
USA	80000.000000000000
Canada	65000.000000000000

Maximum Value

```
select
        country_name,
        'max' as agg_type,
        max(income)
from
        person
inner join
        country
        on person.country_id = country.id
group by
        country_name;
```

country_name	agg_type	avg
USA	max	80000.00
Canada	max	70000.00

Minimum Value

```
select
        country_name,
        'min' as agg_type,
        min(income)
from
        person
inner join
        country
                on person.country_id = country.id
group by
        country_name;
```

country_name	agg_type	avg
USA	min	80000.00
Canada	min	60000.00

Avg, Max, and Min Values:

```
select
        country_name,
        min(income),
        max(income),
        avg(income)
from
        person
inner join
        country
                on person.country_id = country.id
group by
        country_name;
```

country_name	min(income)	max(income)	avg(income)
Canada	60000.00	70000.00	65000.000000000000
USA	80000.00	80000.00	80000.000000000000

Union Clause

Instead of giving the queries one at a time to get the avg, max, and min, we can use a union clause. Union clause is used to combine the results of multiple queries. The rules for using a union clause are as follows (see link):

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

```
select country_name, 'avg' as agg_type, avg(income) from person
inner join country on person.country_id = country.id
group by country_name

UNION

select country_name, 'max' as agg_type, max(income) from person
inner join country on person.country_id = country.id
group by country_name

UNION
select country_name, 'min' as agg_type, min(income) from person
inner join country on person.country_id = country.id
group by country_name;
```

country_name	agg_type	avg
Canada	avg	65000.000000000000
Canada	max	70000.00
Canada	min	60000.00
USA	max	80000.00
USA	avg	80000.000000000000
USA	min	80000.00

Sub-queries

Sub-queries can be used to create intermediary tables that are then joined within the larger dataset. It can help with applying filter logic and performance tuning by filtering the amount of data being processed in a join condition.

```
select * from person
full outer join (
        select * from country
        where id = 2) country_table
on person.country_id = country_table.id;
```

id	person_name	income	country_id	id	country_name
1	Sally	60000.00	1		
2	Bob	70000.00	1		
3	Lucy	80000.00	2	2	USA
4	Bill	75000.00	NULL		

Compare the sub-query to filtering after joining:

```
select * from person
full outer join country
on person.country_id = country.id
where country.id = 2;
```

id	person_name	income	country_id	id	country_name
3	Lucy	80000.00	2	2	USA

When you filter after, it takes the results of the join condition and then applies the filter logic. The sub-query will calculate the sub-query first and then apply the join logic.

Case Statement

```
SELECT person_name, CASE
WHEN income BETWEEN 50000 and 61000 THEN 1
WHEN income BETWEEN 62000 and 71000 THEN 2
WHEN income BETWEEN 72000 and 81000 THEN 3 END as income_category
FROM person;
```

person_name	income_category
Sally	1
Bob	2
Lucy	3
Bill	3

Data type conversion

The following link explains how to perform conversion of data from one format into another: https://www.postgresql.org/docs/current/functions-formatting.html

Often times when working in a big data environment, data is dumped into tables and all the columns are treated as strings. Therefore, if you need to perform math operations or other comparisons, you may not get the correct results.

The most common conversions are string to number or string to date/timestamp. Examples of each are below.

Treat string as a number

If you try to add two strings, it will result in an error:

```
select ('100') + ('100'); -- query fails
```

```
SELECT ('100'::INTEGER) + ('200'::Integer); -- result is successfully returned as 300
SELECT ('10.1'::DECIMAL) + ('12.2':: DECIMAL)' -- result is successfully returned as 22.3
SELECT ('-10.1'::DECIMAL) + ('12.2':: DECIMAL)' -- result is successfully returned as 2.1
```

Treat string as a date

```
select ('jan 01 2021') > ('feb 01 2021'); -- incorrectly results in true because 'j' is larger than 'f'
alphabetically.
```

```
select ('jan 01 2021'::DATE) > ('feb 01 2021'::DATE); -- correctly returns false
select ('jan 01 2021 08:01:45'::TIMESTAMP) > ('jan 01 2021 08:01:46'::TIMESTAMP); -- correctly returns false
select to_date('05$DEC$2000', 'DD$MON$YYYY') -- returns 2000-12-05T00:00:00.000Z
```

Run query against table

```
create table test_data_conv (income_1 varchar(100), income_2 varchar(100), date_1 varchar(100), date_2
varchar(100));

INSERT INTO test_data_conv
(income_1, income_2, date_1, date_2)
values
('10.1', '12.2', 'jan 01 2021', 'feb 01 2021'),
('13.1', '14.2', 'mar 01 2021', 'aug 01 2021');

SELECT sum(income_1::DECIMAL) from test_data_conv; --23.2

select * from test_data_conv where (date_1::DATE) < (date_2::DATE); -- correctly returns both rows.
```

```
ALTER TABLE person ADD column bday text;

UPDATE person
SET bday = '2000-Jan-01'
WHERE person_name = 'Lucy';

UPDATE person
SET bday = '2000-Feb-01'
WHERE person_name = 'Bob';

-- compare the results for the following queries:
select * from person
order by bday::date;

select * from person
order by bday;
```

Performance Tuning

There are ways to optimize performance of queries by reducing the amount of data being processed or by using SQL keywords that are optimized for certain processes. Here are a few examples:

Description	Good	Bad
Select particular columns to speed up performance. Makes a difference if the table has many columns.	`select name, age from person;`	`select * from person;`

Reduce amount of data being processed in join clauses	`select * from`
	
	`(select employee_id, country_id from employee where join_date > '2020-01-01') t1`
	
	`inner join`
	
	`(employee_id, salary from compensation where comp_date > '2020-01-01') t2`
	
	`on t1.employee_id = t2.employee_id;`
		`select * from employee t1`
		
		`inner join compensation t2`
		
		`on t1.employee_id = t2.employee_id`
		
		`where join_date > '2020-01-01'`
		
		`and comp_date > '2020-01-01';`

Inserting, Updating, and Deleting Tables and Data

Create tables

```
-- Create the following two tables for this demo:
CREATE TABLE person ( id INTEGER PRIMARY KEY, person_name VARCHAR(100), income NUMERIC(7,2) );
CREATE TABLE country ( id INTEGER PRIMARY KEY, country_name VARCHAR(100) );
```

For more information on creating tables: https://www.postgresql.org/docs/9.2/sql-createtable.html

Insert Data into the Tables

```
-- Insert data into the 'person' table:
INSERT INTO person (id, person_name, income) VALUES (1, 'Sally', 60000);
INSERT INTO person (id, person_name, income) VALUES (2, 'Bob', 70000);
INSERT INTO person (id, person_name, income) VALUES (3, 'Lucy', 80000);
```

```
INSERT INTO country (id, country_name) VALUES (1, 'Canada'), (2, 'USA'), (3, 'Mexico');
```

Update

Update the person table to have a column for the person's country:

```
ALTER TABLE person ADD column country_id integer;
```

Update the data:

```
UPDATE person
SET country_id = 2
WHERE person_name = 'Lucy';
```

```
UPDATE person
SET country_id = 1
WHERE person_name in ('Sally', 'Bob');
```

Delete Data

The delete clause is used to delete data from a table. Be careful when using the delete clause. If you omit the 'where' clause, you'll end up deleting all data from the table.

```
-- delete row from country table where the ID of the country = 3

delete from country where id = 3;
```

Drop tables

We can drop tables from our database by performing the following command.

```
drop table person;
drop table country;
```

Additional online resources:

Name	Reference
Postgres cheatsheet with common commands	https://www.postgresqltutorial.com/postgresql-cheat-sheet/
Postgres tutorial	https://www.postgresqltutorial.com/
Postgres documentation	https://www.postgresql.org/docs/9.4/

Past Recordings:

SQL Introductory workshop conducted at KCNI Oct 2020:

- **Link**: https://camh.webex.com/camh/ldr.php?RCID=a9ab57ca3058aab99d2d2d8de3361bab
- **pw**: KCNIsql1021

Contacts:

- adeel.ansari@camh.ca