

## Aims

This exercise aims to get you to:

- obtain some hands-on experience of the PostgreSQL server.
- write correct SQL code to query the database.

## Preliminaries

Before following the tutorial, make sure you

- have installed your PostgreSQL server correctly
- have logged into vxdb, and executed commands (manually or automatically) to start the database (c.f., Lab 1 web page).

Just a few reminders:

- Use `--` to add comment to your SQL scripts.
- end your SQL commands with a semicolon.
- string literals are enclosed by single quote (`'`) instead of double quote (`"`).

## Tutorial

(The complete tutorial script can be found at `/home/cs9311/web/24T2/lab/06/pgtutorial.sql`. You can make a copy of the data file into your directory and create a database:)

```
$ cp /home/cs9311/web/24T2/lab/06/pgtutorial.sql ./
$ createdb lab04
$ psql lab04
```

)

1. You can create a new table by specifying the table name, along with all column names and their types:

```
CREATE TABLE weather (
    city          varchar(80),
    temp_lo       int,          -- low temperature
    temp_hi       int,          -- high temperature
    prcp          real,         -- precipitation
    date          date
);
```

You can enter this into `psql` with the line breaks. `psql` will recognize that the command is not terminated until the semicolon.

White space (i.e., spaces, tabs, and newlines) may be used freely in SQL commands. That means you can type the command aligned differently than above, or even all on one line. Two dashes (`--`) introduce comments. Whatever follows them is ignored up to the end of the line. SQL is case insensitive about key words and identifiers, except when identifiers are double-quoted to preserve the case (not done above).

`varchar(80)` specifies a data type that can store arbitrary character strings up to 80 characters in length. `int` is the normal integer type. `real` is a type for storing single precision floating-point numbers. `date` should be self-explanatory. (Yes, the column of type `date` is also named `date`. This may be convenient or confusing --- you choose.)

PostgreSQL supports the standard SQL types `int`, `smallint`, `real`, `double precision`, `char(N)`, `varchar(N)`, `date`, `time`, `timestamp`, and `interval`, as well as other types of general utility and a rich set of geometric types. PostgreSQL can be customized with an arbitrary number of user-defined data types. Consequently, type names are not syntactical key words, except where required to support special cases in the SQL standard.

The second example will store cities and their associated geographical location:

```
CREATE TABLE cities (
    name          varchar(80),
```

```
location      point
);
```

The point type is an example of a PostgreSQL-specific data type.

Finally, it should be mentioned that if you don't need a table any longer or want to recreate it differently you can remove it using the following command:

```
DROP TABLE tablename;
```

## 2. Populating a Table with Rows/Tuples

The INSERT statement is used to populate a table with rows/tuples:

```
INSERT INTO weather VALUES ('San Francisco', 46, 50, 0.25, '1994-11-27');
```

Note that all data types use rather obvious input formats. Constants that are not simple numeric values usually must be surrounded by single quotes ('), as in the example. The date type is actually quite flexible in what it accepts, but for this tutorial we will stick to the unambiguous format shown here. (Also see the functions for date/time data types in the documentation)

The point type requires a coordinate pair as input, as shown here:

```
INSERT INTO cities VALUES ('San Francisco', '(-194.0, 53.0)');
```

The syntax used so far requires you to remember the order of the columns. An alternative syntax allows you to list the columns explicitly:

```
INSERT INTO weather (city, temp_lo, temp_hi, prcp, date)
VALUES ('San Francisco', 43, 57, 0.0, '1994-11-29');
```

You can list the columns in a different order if you wish or even omit some columns, e.g., if the precipitation is unknown:

```
INSERT INTO weather (date, city, temp_hi, temp_lo)
VALUES ('1994-11-29', 'Hayward', 54, 37);
```

Many developers consider explicitly listing the columns better style than relying on the order implicitly.

Please enter all the commands shown above so you have some data to work with in the following sections.

## 3. Querying a Table

To retrieve data from a table, the table is queried. An SQL SELECT statement is used to do this. The statement is divided into a select list (the part that lists the columns to be returned), a table list (the part that lists the tables from which to retrieve the data), and an optional qualification (the part that specifies any restrictions). For example, to retrieve all the rows of table weather, type:

```
SELECT * FROM weather;
```

Here is a shorthand for “all columns”. (While SELECT is useful for off-the-cuff queries, it is widely considered bad style in production code, since adding a column to the table would change the results) So the same result would be had with:

```
SELECT city, temp_lo, temp_hi, prcp, date FROM weather;
```

The output should be:

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	43	57	0	1994-11-29
Hayward	37	54		1994-11-29

(3 rows)

You can write expressions, not just simple column references, in the select list. For example, you can do:

```
SELECT city, (temp_hi+temp_lo)/2 AS temp_avg, date FROM weather;
```

This should give:

city	temp_avg	date
------	----------	------

San Francisco		48		1994-11-27
San Francisco		50		1994-11-29
Hayward		45		1994-11-29
(3 rows)				

Notice how the AS clause is used to relabel the output column. (The AS clause is optional.)

A query can be “qualified” by adding a WHERE clause that specifies which rows are wanted. The WHERE clause contains a Boolean (truth value) expression, and only rows for which the Boolean expression is true are returned. The usual Boolean operators (AND, OR, and NOT) are allowed in the qualification. For example, the following retrieves the weather of San Francisco on rainy days:

```
SELECT * FROM weather
WHERE city = 'San Francisco' AND prcp > 0.0;
```

Result:

city		temp_lo		temp_hi		prcp		date
-----+								
San Francisco		46		50		0.25		1994-11-27
(1 row)								

You can request that the results of a query be returned in sorted order:

```
SELECT * FROM weather
ORDER BY city;
```

city		temp_lo		temp_hi		prcp		date
-----+								
Hayward		37		54				1994-11-29
San Francisco		43		57		0		1994-11-29
San Francisco		46		50		0.25		1994-11-27

In this example, the sort order isn't fully specified, and so you might get the San Francisco rows in either order. But you'd always get the results shown above if you do

```
SELECT * FROM weather
ORDER BY city, temp_lo;
```

You can request that duplicate rows be removed from the result of a query:

```
SELECT DISTINCT city
FROM weather;
```

city
-----
Hayward
San Francisco
(2 rows)

Here again, the result row ordering might vary. You can ensure consistent results by using DISTINCT and ORDER BY together:

```
SELECT DISTINCT city
FROM weather
ORDER BY city;
```

#### 4. Joins between Tables

Thus far, our queries have only accessed one table at a time. Queries can access multiple tables at once, or access the same table in such a way that multiple rows of the table are being processed at the same time. A query that accesses multiple rows of the same or different tables at one time is called a join query. As an example, say you wish to list all the weather records together with the location of the associated city. To do that, we need to compare the city column of each row of the weather table with the name column of all rows in the cities table, and select the pairs of rows where these values match.

**Note:** This is only a conceptual model. The join is usually performed in a more efficient manner than actually comparing each possible pair of rows, but this is invisible to the user.

This would be accomplished by the following query:

```
SELECT *
FROM weather, cities
WHERE city = name;
```

city	temp_lo	temp_hi	prcp	date	name	location
San Francisco	46	50	0.25	1994-11-27	San Francisco	(-194,53)
San Francisco	43	57	0	1994-11-29	San Francisco	(-194,53)

(2 rows)

Observe two things about the result set:

- There is no result row for the city of Hayward. This is because there is no matching entry in the `cities` table for Hayward, so the join ignores the unmatched rows in the `weather` table. We will see shortly how this can be fixed.
- There are two columns containing the city name. This is correct because the lists of columns of the `weather` and the `cities` table are concatenated. In practice this is undesirable, though, so you will probably want to list the output columns explicitly rather than using `*`:

```
SELECT city, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;
```

*\*Exercise\*:* Attempt to find out the semantics of this query when the `WHERE` clause is omitted.

Since the columns all had different names, the parser automatically found out which table they belong to. If there were duplicate column names in the two tables you'd need to qualify the column names to show which one you meant, as in:

```
SELECT weather.city, weather.temp_lo, weather.temp_hi,
       weather.prcp, weather.date, cities.location
FROM weather, cities
WHERE cities.name = weather.city;
```

It is widely considered good style to qualify all column names in a join query, so that the query won't fail if a duplicate column name is later added to one of the tables.

Join queries of the kind seen thus far can also be written in this alternative form:

```
SELECT *
FROM weather INNER JOIN cities ON (weather.city = cities.name);
```

This syntax is not as commonly used as the one above, but we show it here to help you understand the following topics.

Now we will figure out how we can get the Hayward records back in. What we want the query to do is to scan the `weather` table and for each row to find the matching `cities` row(s). If no matching row is found we want some "empty values" to be substituted for the `cities` table's columns. This kind of query is called an `_outer join`. (*The joins we have seen so far are inner joins.*) The command looks like this:

```
SELECT *
FROM weather LEFT OUTER JOIN cities ON (weather.city = cities.name);
```

city	temp_lo	temp_hi	prcp	date	name	location
Hayward	37	54		1994-11-29		
San Francisco	46	50	0.25	1994-11-27	San Francisco	(-194,53)
San Francisco	43	57	0	1994-11-29	San Francisco	(-194,53)

(3 rows)

This query is called a `_left outer join` because the table mentioned on the left of the join operator will have each of its rows in the output at least once, whereas the table on the right will only have those rows output that match some row of the left table. When outputting a left-table row for which there is no right-table match, empty (null) values are substituted for the right-table columns.

**Exercise:** There are also right outer joins and full outer joins. Try to find out what those do.

We can also join a table against itself. This is called a *self join*. As an example, suppose we wish to find all the weather records that are in the temperature range of other weather records. So we need to compare the `temp_lo` and `temp_hi` columns of each weather row to the `tempLo` and `tempHi` columns of all other weather rows. We can do this with the following query:

```
SELECT W1.city, W1.temp_lo AS low, W1.temp_hi AS high,
       W2.city, W2.temp_lo AS low, W2.temp_hi AS high
FROM weather W1, weather W2
WHERE W1.temp_lo < W2.temp_lo
AND W1.temp_hi > W2.temp_hi;
```

city	low	high	city	low	high
San Francisco	43	57	San Francisco	46	50
Hayward	37	54	San Francisco	46	50

(2 rows)

Here we have relabeled the weather table as W1 and W2 to be able to distinguish the left and right side of the join. You can also use these kinds of aliases in other queries to save some typing, e.g.:

```
SELECT *
FROM weather w, cities c
WHERE w.city = c.name;
```

You will encounter this style of abbreviating quite frequently.

## 5. Aggregate Functions

Like most other relational database products, PostgreSQL supports aggregate functions. An aggregate function computes a single result from multiple input rows. For example, there are aggregates to compute the count, sum, avg (average), max (maximum) and min (minimum) over a set of rows.

As an example, we can find the highest low-temperature reading anywhere with

```
SELECT max(temp_lo) FROM weather;
```

max
-----
46

(1 row)

If we wanted to know what city (or cities) that reading occurred in, we might try

```
SELECT city FROM weather WHERE temp_lo = max(temp_lo);    -- WRONG
```

but this will not work since the aggregate max cannot be used in the WHERE clause. (This restriction exists because the WHERE clause determines which rows will be included in the aggregate calculation; so obviously it has to be evaluated before aggregate functions are computed.) However, as is often the case the query can be restated to accomplish the desired result, here by using a **subquery**:

```
SELECT city FROM weather
WHERE temp_lo = (SELECT max(temp_lo) FROM weather);
```

city
-----
San Francisco

(1 row)

This is OK because the subquery is an independent computation that computes its own aggregate separately from what is happening in the outer query.

Aggregates are also very useful in combination with GROUP BY clauses. For example, we can get the maximum low temperature observed in each city with

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city;
```

city	max
Hayward	37
San Francisco	46

(2 rows)

which gives us one output row per city. Each aggregate result is computed over the table rows matching that city. We can filter these grouped rows using HAVING:

```
SELECT city, max(temp_lo)
FROM weather
GROUP BY city
HAVING max(temp_lo) < 40;
```

city	max
Hayward	37

(1 row)

which gives us the same results for only the cities that have all temp\_lo values below 40. Finally, if we only care about cities whose names begin with "S", we might do

```
SELECT city, max(temp_lo)
FROM weather
WHERE city LIKE 'S%'
GROUP BY city
HAVING max(temp_lo) < 40;
```

(The LIKE operator does pattern matching.)

It is important to understand the interaction between aggregates and SQL's WHERE and HAVING clauses. The fundamental difference between WHERE and HAVING is this: WHERE selects input **rows** *before* groups and aggregates are computed (thus, it controls which rows go into the aggregate computation), whereas HAVING selects **group** rows *after* groups and aggregates are computed. Thus, the WHERE clause must not contain aggregate functions; it makes no sense to try to use an aggregate to determine which rows will be inputs to the aggregates. On the other hand, the HAVING clause always contains aggregate functions. (Strictly speaking, you are allowed to write a HAVING clause that doesn't use aggregates, but it's seldom useful. The same condition could be used more efficiently at the WHERE stage.)

In the previous example, we can apply the city name restriction in WHERE, since it needs no aggregate. This is more efficient than adding the restriction to HAVING, because we avoid doing the grouping and aggregate calculations for all rows that fail the WHERE check.

## 6. Updates

You can update existing rows using the UPDATE command. Suppose you discover the temperature readings are all off by 2 degrees after November 28. You may correct the data as follows:

```
UPDATE weather
SET temp_hi = temp_hi - 2, temp_lo = temp_lo - 2
WHERE date > '1994-11-28';
```

Look at the new state of the data:

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29
Hayward	35	52		1994-11-29

(3 rows)

## 7. Deletions

Rows can be removed from a table using the DELETE command. Suppose you are no longer interested in the weather of Hayward. Then you can do the following to delete those rows from the table:

```
DELETE FROM weather WHERE city = 'Hayward';
```

All weather records belonging to Hayward are removed.

```
SELECT * FROM weather;
```

city	temp_lo	temp_hi	prcp	date
San Francisco	46	50	0.25	1994-11-27
San Francisco	41	55	0	1994-11-29

(2 rows)

One should be wary of statements of the form

```
DELETE FROM tablename;
```

Without a qualification, DELETE will remove all rows from the given table, leaving it empty. The system will not request confirmation before doing this!

## 8. Foreign Keys

Consider the following problem: You want to make sure that no one can insert rows in the weather table that do not have a matching entry in the cities table. This is called maintaining the *referential integrity* of your data. In simplistic database systems this would be implemented (if at all) by first looking at the cities table to check if a matching record exists, and then inserting or rejecting the new weather records. This approach has a number of problems and is very inconvenient, so PostgreSQL can do this for you by adding the referential integrity constraint into the CREATE TABLE statements.

The new declaration of the tables would look like this:

```
CREATE TABLE cities (
    name      varchar(80) primary key,
    location  point
);

CREATE TABLE weather (
    city      varchar(80) references cities(name),
    temp_lo   int,
    temp_hi   int,
    prcp      real,
    date      date
);
```

Now try inserting an invalid record:

```
INSERT INTO weather VALUES ('Berkeley', 45, 53, 0.0, '1994-11-28');

ERROR:  insert or update on table "weather" violates foreign key constraint "weather_city_fkey"
DETAIL:  Key (city)=(Berkeley) is not present in table "cities".
```

The behavior of foreign keys can be finely tuned to your application. We will not go beyond this simple example in this tutorial, but just refer you to Chapter 5 in PostgreSQL documentation for more information. Making correct use of foreign keys will definitely improve the quality of your database applications, so you are strongly encouraged to learn about them.

## 9. Views

Suppose the combined listing of weather records and city location is of particular interest to your application, but you do not want to type the join query each time you need it. You can create a view over the query, which gives a name to the query that you can refer to like an ordinary table.

```
CREATE VIEW myview AS
SELECT city, temp_lo, temp_hi, prcp, date, location
FROM weather, cities
WHERE city = name;

SELECT * FROM myview;
```

Making liberal use of views is a key aspect of good SQL database design. Views allow you to encapsulate the details of the structure of your tables, which may change as your application evolves, behind consistent interfaces.

Views can be used in almost any place a real table can be used. Building views upon other views is not uncommon.