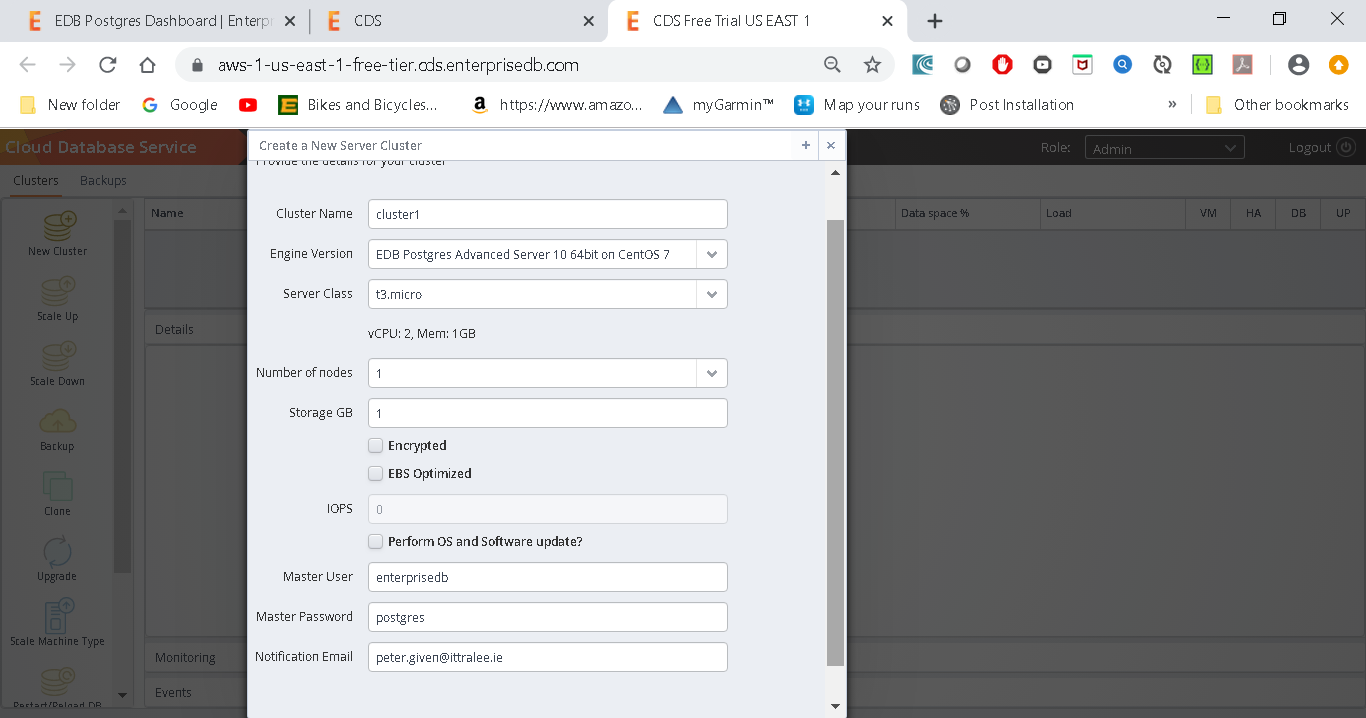
PostgreSQL Lab

You can install PostgreSQL in many ways, depending on your operating system – you can install the latest windows version (http://www.enterprisedb.com/products-services-training/pgdownload#windows)

Or try the free cloud instance (you still need to download the local version to get a client to log on with)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_



Default everything else

**Connecting with the psql Client**

To use the psql client to connect to an instance, navigate into the directory in which the psql client is installed on your local system and enter:

./psql -h <instance\_address> -p 9999 -U <superuser\_name> -d <database\_name>

Where

*instance\_address* is the address of the master node of the cluster.

*superuser\_name* is the name of the database superuser.

*database\_name* is the name of the database to which you will connect.

When prompted, provide the password associated with the CDS instance.

For example:

$ ./psql -h ec2-3-93-214-74.compute-1.amazonaws.com -p 9999 -U

enterprisedb

-d edb

Password **for** user enterprisedb:

psql.bin (11.0, server 11.1.7)

SSL connection (protocol: TLSv1.2, cipher: AES256-GCM-SHA384, bits: 256, compression: off)

Type "help" **for** help.

edb=*#*

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Start the pgAdmin application

And connect with your password (username postgres)

Now open the PosgreSQL shell

Once you have Postgres installed, create a schema called book using the following command:

**create database book;**

\c book

To see extensions

\dx

**Add extensions**

CREATE EXTENSION tablefunc;

CREATE EXTENSION dict\_xsyn;

CREATE EXTENSION fuzzystrmatch;

CREATE EXTENSION pg\_trgm;

CREATE EXTENSION cube;

**CRUD with PostgreSQL (including referential integrity)**

**CREATE TABLE** countries ( country\_code **char**(2) **PRIMARY KEY**, country\_name **text UNIQUE** );

**INSERT INTO** countries (country\_code, country\_name) **VALUES** (*'us'*,*'United States'*), (*'mx'*,*'Mexico'*), (*'au'*,*'Australia'*), (*'gb'*,*'United Kingdom'*), (*'de'*,*'Germany'*), (*'ll'*,*'Loompaland'*);

**SELECT** \* **FROM** countries;

**DELETE FROM** countries **WHERE** country\_code = *'ll'*;

**CREATE TABLE** cities ( name **text NOT** NULL, postal\_code **varchar**(9) CHECK (postal\_code <> *''*), country\_code **char**(2) REFERENCES countries, **PRIMARY KEY** (country\_code, postal\_code) );

**INSERT INTO** cities **VALUES** (*'Toronto'*,*'M4C1B5'*,*'ca'*);

**INSERT INTO** cities **VALUES** (*'Portland'*,*'87200'*,*'us'*);

**UPDATE** cities **SET** postal\_code = *'97205'* **WHERE** name = *'Portland'*;

**Table Joins**

The basic form of a join is the *inner join*. In the simplest form, you specify two columns (one from each table) to match by, using the ON keyword.

**SELECT** cities.\*, country\_name **FROM** cities **INNER JOIN** countries **ON** cities.country\_code = countries.country\_code;

We can also join a table like cities that has a compound primary key. To test a compound join, let’s create a new table that stores a list of venues.

**CREATE TABLE** venues (venue\_id SERIAL **PRIMARY KEY**,name **varchar**(255),street\_address **text**,type **char**(7) CHECK ( type in (*'public'*,*'private'*) ) **DEFAULT** *'public'*, postal\_code **varchar**(9),country\_code **char**(2),FOREIGN **KEY** (country\_code,postal\_code) REFERENCES cities (country\_code, postal\_code) MATCH FULL);

**INSERT INTO** venues (name, postal\_code, country\_code) **VALUES** (*'Crystal Ballroom'*, *'97205'*, *'us'*);

Joining the venues table with the cities table requires *both* foreign key columns.

To save on typing, we can alias the table names by following the real table name directly with an alias, with an optional AS between (for example, venues v or venues AS v).

**SELECT** v.venue\_id, v.name, c.name **FROM** venues v **INNER JOIN** cities c **ON** v.postal\_code=c.postal\_code **AND** v.country\_code=c.country\_code;

You can optionally request that PostgreSQL return columns after insertion by ending the query with a RETURNING statement.

**INSERT INTO** venues (name, postal\_code, country\_code) **VALUES** (*'Voodoo Donuts'*, *'97205'*, *'us'*) RETURNING venue\_id;

Outer Joins

In addition to inner joins, PostgreSQL can also perform *outer joins*.

Outer joins are a way of merging two tables when the results of one table must always be returned, whether or not any matching column values exist on the other table.

It’s easiest to give an example, but to do that, we’ll create a new table named events.

 Your events table should have these columns: a SERIAL integer event\_id, a title, starts and ends (of type *timestamp*), and a venue\_id (foreign key that references venues). See the screen dump below.

create table events (event\_id SERIAL PRIMARY KEY, title VARCHAR(30), starts timestamp, ends timestamp, venue\_id SERIAL REFERENCES venues);

After creating the events table, INSERT the following values (timestamps are inserted as a string like *2012-02-15 17:30*), two holidays, and a club

**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (*'LARP Club'*, *'2020-02-15 17:30'*, *'2020-02-15 19:30', 2);*

**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (*'April Fools Day'*, *'2020-04-01 17:30'*, *'2020-04-01 19:30', 1);*

**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (*'Christmas Day'*, *'2020-12-25 17:30'*, *'2020-12-25 19:30', 2);*

Let’s first craft a query that returns an event title and venue name as an inner join (the word INNER from INNER JOIN is not required, so leave it off here).

**SELECT** e.title, v.name **FROM** events e **JOIN** venues v **ON** e.venue\_id = v.venue\_id;

INNER JOIN will return a row only *if the column values match*.

**INSERT INTO** venues (name, postal\_code, country\_code) **VALUES** (*'Horans'*, *'97205'*, *'us'*) RETURNING venue\_id;

Retrieving all of the venue, whether or not they have a event, requires a LEFT OUTER JOIN (shortened to LEFT JOIN).

**SELECT** v.name, e.title **FROM** venues v **LEFT JOIN** events e **ON** v.venue\_id = e.venue\_id;

If you require the inverse, all venues and only matching events, use a RIGHT JOIN.

Finally, there’s the FULL JOIN, which is the union of LEFT and RIGHT; you’re guaranteed all values from each table, joined wherever columns match.

**Indexes**

**CREATE INDEX** events\_title **ON** events **USING** hash (title);

For less-than/greater-than/equals-to matches, we want an index more flexible than a simple hash, like a B-tree. Consider a query to find all events that are on or after April 1.

**SELECT** \* **FROM** events **WHERE** starts >= *'2020-04-01'*;

For this, a tree is the perfect data structure. To index the starts column with a B-tree, use this:

**CREATE INDEX** events\_starts **ON** events **USING** btree (starts);

\di

**Stored Procedures**

Let’s create a procedure (or FUNCTION) that simplifies INSERTing a new event at a venue without needing the venue\_id

If the venue doesn’t exist, create it first and reference it in the new event. Also, we’ll return a boolean indicating whether a new venue was added, as a nicety to our users.

**CREATE OR REPLACE** FUNCTION add\_event( title **text**, starts **timestamp**, ends **timestamp**, venue **text**, postal **varchar**(9), country **char**(2) ) RETURNS **boolean AS**

$$ DECLARE did\_insert **boolean** := false;

found\_count **integer**;

the\_venue\_id **integer**;

**BEGIN**

**SELECT** venue\_id **INTO** the\_venue\_id **FROM** venues v **WHERE** v.postal\_code=postal **AND** v.country\_code=country **AND** v.name ILIKE venue LIMIT 1;

**IF** the\_venue\_id IS NULL **THEN** **INSERT INTO** venues (name, postal\_code, country\_code) **VALUES** (venue, postal, country) RETURNING venue\_id **INTO** the\_venue\_id;

did\_insert := true;

**END IF**;

*-- Note: not an “error”, as in some programming languages*

RAISE NOTICE *'Venue found %'*, the\_venue\_id;

**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (title, starts, ends, the\_venue\_id);

RETURN did\_insert;

**END**;

$$ LANGUAGE plpgsql;

Running it

SELECT add\_event(*'House Party'*, *'2020-05-03 23:00'*, *'2020-05-04 02:00'*, *'Run''s House'*, *'97205'*, *'us'*);

This should return t (true), since this is the first use of the venue *Run’s House*.

This saves a client two round-trip SQL commands to the database (a select and then an insert) and instead does only one.

The language we used in the procedure we wrote is PL/pgSQL (which stands for Procedural Language/PostgreSQL).

 In addition to PL/pgSQL, Postgres supports three more core languages for writing procedures: Tcl, Perl, and Python.

People have written extensions for a dozen more including Ruby, Java, PHP, Scheme, and others listed in the public documentation.

Triggers

Triggers automatically fire stored procedures when some event happens, like an insert or update.

They allow the database to enforce some required behaviour in response to changing data.

Let’s create a new PL/pgSQL function that logs whenever an event is updated (we want to be sure no one changes an event and tries to deny it later).

First, create a logs table to store event changes. A primary key isn’t necessary here, since it’s just a log.

**CREATE TABLE** logs ( event\_id **integer**, old\_title **varchar**(255), old\_starts **timestamp**, old\_ends **timestamp**, logged\_at **timestamp DEFAULT** current\_timestamp );

Next, we build a function to insert old data into the log. The OLD variable represents the row about to be changed (NEW represents an incoming row, which we’ll see in action soon enough). Output a notice to the console with the event\_id before returning.

**CREATE OR REPLACE** FUNCTION log\_event() RETURNS **trigger AS**

$$ DECLARE

**BEGIN**

**INSERT INTO** logs (event\_id, old\_title, old\_starts, old\_ends) **VALUES** (OLD.event\_id, OLD.title, OLD.starts, OLD.ends);

RAISE NOTICE *'Someone just changed event #%'*, OLD.event\_id; RETURN NEW;

**END**;

$$ LANGUAGE plpgsql;

\df

Finally, we create our trigger to log changes after any row is updated.

**CREATE TRIGGER** log\_events AFTER **UPDATE ON** events FOR EACH ROW EXECUTE PROCEDURE log\_event();

So, it turns out our party at Run’s House has to end earlier than we hoped. Let’s change the event.

**UPDATE** events **SET** ends=*'2020-05-04 01:00:00'* **WHERE** title=*'House Party'*;

To see the resulting log record

**SELECT** event\_id, old\_title, old\_ends, logged\_at **FROM** logs;

**Views**

**CREATE OR REPLACE** VIEW holidays **AS** **SELECT** event\_id **AS** holiday\_id, title **AS** name, starts **AS date FROM** events **WHERE** title **LIKE** *'%Day%'* ;

\dv

Now you can query holidays like any other table. Under the covers it’s the plain old events table.

As proof, add *Valentine’s Day* on *2020- 02-14* to events

**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (*'Valentines Day'*, *'2020-02-14 17:30'*, *'2020-02-14 19:30', 2);*

and query the holidays view.

**SELECT** name, to\_char(**date**, *'Month DD, YYYY'*) **AS date FROM** holidays **WHERE date** <= *'2020-04-01'*;

**Alter Tables**

If you want to add a new column to the view, it will have to come from the underlying table. Let’s alter the events table to have an array of associated colors.

ALTER **TABLE** events ADD colors **text** ARRAY;

Since holidays are to have colors associated with them, let’s update the VIEW query to contain the colors array.

**CREATE OR REPLACE** VIEW holidays **AS** **SELECT** event\_id **AS** holiday\_id, title **AS** name, starts **AS date**, colors **FROM** events **WHERE** title **LIKE** *'%Day%'*;

Now it’s a matter of setting an array or color strings to the holiday of choice.

UPDATE holidays SET colors = *'{"red","green"}'* where name = *'Christmas Day'*;

Unfortunately, we cannot update a view directly.

Looks like we need a RULE.

A RULE is a description of how to alter the parsed *query tree*.

What’s more is that a VIEW such as holidays *is* a RULE.

We can prove this by taking a look at the execution plan of the holidays view using the EXPLAIN command (notice *Filter* is the WHERE clause, and *Output* is the column list).

EXPLAIN VERBOSE **SELECT** \* **FROM** holidays;

Compare that to running EXPLAIN VERBOSE on the query we built the holidays VIEW from. They’re functionally identical.

EXPLAIN VERBOSE **SELECT** event\_id **AS** holiday\_id, title **AS** name, starts **AS date**, colors **FROM** events **WHERE** title **LIKE** *'%Day%'*;

So, to allow updates against our holidays view, we need to craft a RULE that tells Postgres what to do with an UPDATE.

Our rule will capture updates to the holidays view and instead run the update on events, pulling values from the pseudorelations NEW and OLD.

NEW functionally acts as the relation containing the values we’re setting, while OLD contains the values we query by.

**CREATE** RULE update\_holidays **AS ON UPDATE** TO holidays DO INSTEAD **UPDATE** events **SET** title = NEW.name, starts = NEW.**date**, colors = NEW.colors **WHERE** title = OLD.name;

With this rule in place, now we can update holidays directly.

**UPDATE** holidays **SET** colors = *'{"red","green"}'* **where** name = *'Christmas Day'*;

Next let’s insert *New Years Day* on *2020-01-01* into holidays. As expected, we need a rule for that too. No problem.

**CREATE** RULE insert\_holidays **AS ON INSERT** TO holidays DO INSTEAD **INSERT INTO events**

Crosstab

For our last exercise , we’re going to build a monthly calendar of events, where each month in the calendar year counts the number of events in that month.

This kind of operation is commonly done by a *pivot table*.

These constructs “pivot” grouped data around some other output, in our case, a list of months. We’ll build our pivot table using the crosstab() function.

Start by crafting a query to count the number of events per month, each year.

PostgreSQL provides an extract() function that returns some subfield from a date or timestamp, which aids in our grouping.

**SELECT** extract(**year from** starts) **as year**, extract(month **from** starts) **as** month, count(\*) **FROM** events **GROUP BY year**, month;

To use crosstab(), the query must return three columns: rowid, category, and value.

We’ll be using the year as an ID, which means the other fields are category (the month) and value (the count).

The crosstab() function needs another set of values to represent months.

This is how the function knows how many columns we need. These are the values that become the columns (the table to *pivot* against).

So, let’s create a table to store a temporary list of numbers.

**CREATE** TEMPORARY **TABLE** month\_count(month **INT**);

**INSERT INTO** month\_count **VALUES** (1),(2),(3),(4),(5),(6),(7),(8),(9),(10),(11),(12);

Now we’re ready to call crosstab() with our two queries.

**SELECT** \* **FROM** crosstab( *'SELECT extract(year from starts) as year,*  *extract(month from starts) as month, count(\*) FROM events* *GROUP BY year, month'*,  *'SELECT \* FROM month\_count'* );

An error occurred.

It may feel cryptic, but it’s saying the function is returning a set of records (rows), but it doesn’t know how to label them. In fact, it doesn’t even know what datatypes they are.

Remember, the pivot table is using our months as categories, but those months are just integers. So, we define them like this:

**SELECT** \* **FROM** crosstab( *'SELECT extract(year from starts) as year,*  *extract(month from starts) as month, count(\*) FROM events* *GROUP BY year, month'*,  *'SELECT \* FROM month\_count'* ) **AS** ( **year int**, jan **int**, feb **int**, mar **int**, apr **int**, may **int**, jun **int**, jul **int**, aug **int**, sep **int**, **oct int**, nov **int**, dec **int** ) **ORDER BY YEAR**;

We have one column year (which is the row ID) and twelve more columns representing the months.

Full-Text and Multidimensions

**CREATE TABLE** genres ( name **text UNIQUE**, position **integer**  );

**CREATE TABLE** movies ( movie\_id SERIAL **PRIMARY KEY**, title **text**, genre cube );

**CREATE TABLE** actors ( actor\_id SERIAL **PRIMARY KEY**, name **text**  );

**CREATE TABLE** movies\_actors ( movie\_id **integer** REFERENCES movies **NOT** NULL, actor\_id **integer** REFERENCES actors **NOT** NULL, **UNIQUE** (movie\_id, actor\_id) );

**CREATE INDEX** movies\_actors\_movie\_id **ON** movies\_actors (movie\_id);

**CREATE INDEX** movies\_actors\_actor\_id **ON** movies\_actors (actor\_id);

**CREATE INDEX** movies\_genres\_cube **ON** movies **USING** gist (genre);

You can download the movies\_data.sql file as a file from the X drive and populate the tables by piping the file into the database.

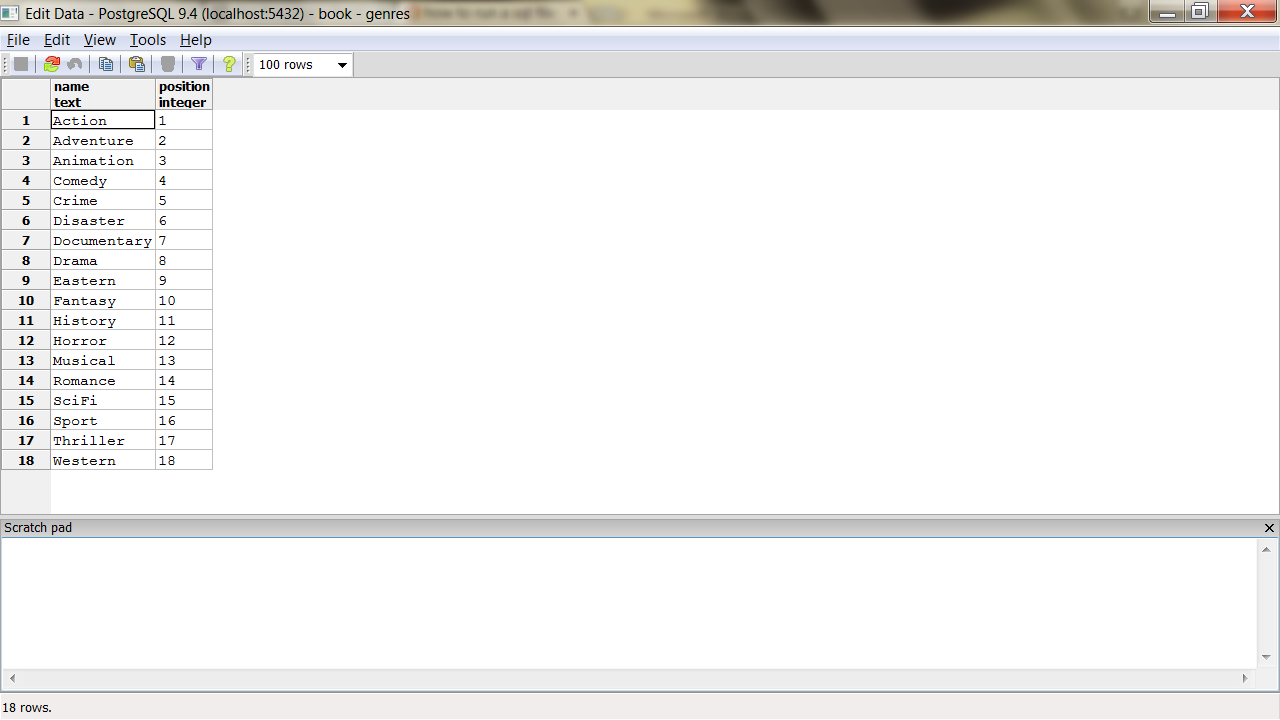
Started pgAdmin and connected to the book database

Select the query tool from tools

Open the movies\_data.sql file

Run the query using Execute pgScript (or Execute?)

Now I could see the data in pgAdmin:

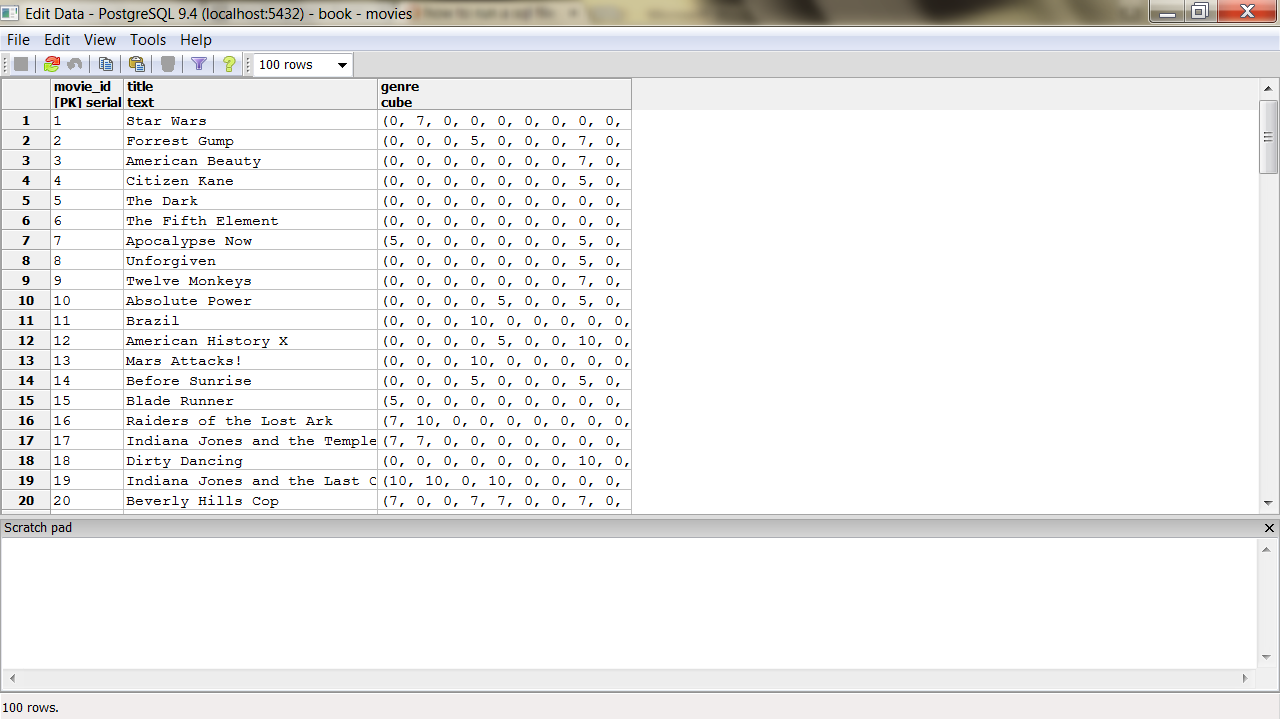


Or

Select \* from genres;

Also view Movies

Select \* from movies;



SQL Standard String Matches

PostgreSQL has many ways of performing text matches, but the two big default methods are LIKE and regular expressions.

LIKE and ILIKE

LIKE and ILIKE (case-insensitive LIKE) are the simplest forms of text search.

They are fairly universal in relational databases. LIKE compares column values against a given pattern string.

The % and \_ characters are wildcards. % matches any number of any characters, and \_ matches exactly one character.

**SELECT** title **FROM** movies **WHERE** title ILIKE *'stardust%'*;

If we want to be sure the substring *stardust* is not at the end of the string, we can use the underscore (\_) character as a little trick.

**SELECT** title **FROM** movies **WHERE** title ILIKE *'stardust\_%'*;

Regex

A more powerful string-matching syntax is a *regular expression* (regex).

Postgres conforms (mostly) to the POSIX style.

In Postgres, a regular expression match is led by the ~ operator, with the optional ! (meaning, *not* matching) and \* (meaning *case insensitive*).

So, to count all movies that do *not* begin with *the*, the following case-insensitive query will work. The characters inside the string are the regular expression.

SELECT COUNT(\*) FROM movies WHERE title !~\* *'^the.\*'*;

You can index strings for pattern matching the previous queries by creating a text\_pattern\_ops operator class index, as long as the values are indexed in lowercase.

**CREATE INDEX** movies\_title\_pattern **ON** movies (lower(title) text\_pattern\_ops);

If you need to index varchars, chars, or names, use the related ops: varchar\_pattern\_ops, bpchar\_pat- tern\_ops, and name\_pattern\_ops.

Levenshtein

Levenshtein is a string comparison algorithm that compares how similar two strings are by how many *steps* are required to change one string into another.

Each replaced, missing, or added character counts as a step.

The distance is the total number of steps away.

In PostgreSQL, the levenshtein() function is provided by the fuzzystrmatch contrib package. Say we have the string *bat* and the string *fads*.

**SELECT** levenshtein(*'bat'*, *'fads'*);

The Levenshtein distance is 3 because—compared to the string *bat*—we replaced two letters (b=>f, t=>d), and we added a letter (+s).

Each change increments the distance.

Changes in case cost a point too, so you may find it best to convert all strings to the same case when querying.

**SELECT** movie\_id, title **FROM** movies **WHERE** levenshtein(lower(title), lower(*'a hard day nght'*)) <= 3;

Trigram

A trigram is a group of three consecutive characters taken from a string.

The pg\_trgm contrib module breaks a string into as many trigrams as it can.

**SELECT** show\_trgm(*'Avatar'*);

Finding a matching string is as simple as counting the number of matching trigrams.

The strings with the most matches are the most similar.

It’s useful for doing a search where you’re OK with either slight misspellings or even minor words missing

The longer the string, the more trigrams and the more likely a match—they’re great for something like movie titles, since they have relatively similar lengths.

We’ll create a trigram index against movie names to start (we use Generalized Index Search Tree [GIST], a generic index API made available by the PostgreSQL engine).

**CREATE INDEX** movies\_title\_trigram **ON** movies **USING** gist (title gist\_trgm\_ops);

Now you can query with a few misspellings and still get decent results.

**SELECT** \* **FROM** movies **WHERE** title % *'Avatre'*;

Full-Text Search

Next, we want to allow users to perform full-text searches based on matching words, even if they’re pluralized.

If a user wants to search for certain words in a movie title but can remember only some of them, Postgres supports simple natural-language processing.

TSVector and TSQuery

Let’s look for a movie that contains the words *night* and *day*. This is a perfect job for text search using the @@ full-text query operator.

**SELECT** title **FROM** movies **WHERE** title @@ *'night & day'*;

The query returns titles like *A Hard Day’s Night*, despite the word *Day* being in possessive form, and the two words are out of order in the query.

The @@ operator converts the name field into a tsvector and converts the query into a tsquery.

A tsvector is a datatype that splits a string into an array (or a *vector*) of tokens, which are searched against the given query, while the tsquery represents a query in some language, like English or French.

You can take a look at how the vector and the query break apart the values by running the conversion functions on the strings outright.

**SELECT** to\_tsvector(*'A Hard Day''s Night'*), to\_tsquery(*'english'*, *'night & day'*);

The tokens on a tsvector are called *lexemes* and are coupled with their positions in the given phrase.

You may have noticed the tsvector for *A Hard Day’s Night* did not contain the lexeme *a*. Moreover, simple English words like *a* are missing if you try to query by them.

**SELECT** \* **FROM** movies **WHERE** title @@ to\_tsquery(*english, 'a'*);

Common words like *a* are called *stop words* and are generally not useful for performing queries.

Indexing Lexemes

Full-text search is powerful. But if we don’t index our tables, it’s also slow.

The EXPLAIN command is a powerful tool for digging into how queries are internally planned.

EXPLAIN **SELECT** \* **FROM** movies **WHERE** title @@ *'night & day'*;

Note the line *Seq Scan on movies*. That’s rarely a good sign in a query, because it means a whole table scan is taking place; each row will be read. So, we need the right index.

We’ll use Generalized Inverted iNdex (GIN)—like GIST, it’s an index API—to create an index of lexeme values we can query against.

The term *inverted index* may sound familiar to you if you’ve ever used a search engine like Lucene or Sphinx.

It’s a common data structure to index full-text searches.

**CREATE INDEX** movies\_title\_searchable **ON** movies **USING** gin(to\_tsvector(*'english'*, title));

With our index in place, let’s try to search again.

EXPLAIN **SELECT** \* **FROM** movies **WHERE** title @@ *'night & day'*;

What happened? Nothing. The index is there, but Postgres isn’t using it.

It’s because our GIN index specifically uses the english configuration for building its tsvectors, but we aren’t specifying that vector.

We need to specify it in the WHERE clause of the query.

EXPLAIN **SELECT** \* **FROM** movies **WHERE** to\_tsvector(*'english'*,title) @@ *'night & day'*;

Metaphones

Are algorithms for creating a string representation of word sounds.

You can define how many characters are in the output string.

For example, the seven-character metaphone of the name Aaron Eck- hart is *ARNKHRT*.

To find all films acted by someone sounding like Broos Wils, we can query against the metaphone output.

Note that NATURAL JOIN is an INNER JOIN that automatically joins ON matching column names (for example, movies.actor\_id= movies\_actors.actor\_id).

**SELECT** title **FROM** movies NATURAL **JOIN** movies\_actors NATURAL **JOIN** actors **WHERE** metaphone(name, 6) = metaphone(*'Broos Wils'*, 6);

The *fuzzystrmatch* module contains other functions: dmetaphone() (double metaphone), dmetaphone\_alt() (for alternative name pronunciations), and soundex() (a really old algorithm from the 1880s made by the U.S. Census to compare common American surnames).

You can dissect the functions’ representations by selecting their output.

( the next query worked best in pgAdmin)

**SELECT** name, dmetaphone(name), dmetaphone\_alt(name), metaphone(name, 8), soundex(name) from Actors

There is no single best function to choose, and the optimal choice depends on your dataset.

Combining String Matches

For example, we could use the trigram operator against metaphone() outputs and then order the results by the lowest Levenshtein distance.

This means “Get me names that sound the most like Robin Williams, in order.”

**SELECT** \* **FROM** actors **WHERE** metaphone(name,8) % metaphone(*'Robin Williams'*,8) **ORDER BY** levenshtein(lower(*'Robin Williams'*), lower(name));

Note it isn’t perfect. Robin Williams ranked at #3. Unbridled exploitation of this flexibility can yield other funny results, so be careful.

**SELECT** \* **FROM** actors **WHERE** dmetaphone(name) % dmetaphone(*'Ron'*);

Genres as a Multidimensional Hypercube

We’ll use the cube datatype to map a movie’s genres as a multidimensional vector.

We will then use methods to efficiently query for the closest points within the boundary of a hypercube to give us a list of similar movies.

Earlier in the movies table we created a column named genres of type cube.

Each value is a point in 18-dimensional space with each dimension representing a genre.

In our system, each genre is scored from (the totally arbitrary numbers) 0 to 10 based on how strong the movie is within that genre—with 0 being nonexistent and 10 being the strongest.

*Star Wars* has a genre vector of (0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0).

We can decrypt its genre values by extracting the cube\_ur\_coord(vector,dimension) using each genres.position. For clarity, we filter out genres with scores of 0.

**SELECT** name, cube\_ur\_coord(*'(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'*, position) **as** score **FROM** genres g **WHERE** cube\_ur\_coord(*'(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'*, position) > 0;

We will find similar movies by finding the nearest points.

The nearest matches to the genre vector can be discovered by the cube\_distance(point1, point2). Here we can find the distance of all movies to the *Star Wars* genre vector, nearest first.

**SELECT** \*, cube\_distance(genre, *'(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'*) dist **FROM** movies **ORDER BY** dist;

We created the movies\_genres\_cube cube index earlier when we created the tables.

However, even with an index, this query is still relatively slow, since it requires a full-table scan. It computes the distance on every row and then sorts them.

Rather than compute the distance of every point, we can instead focus on likely points by way of a *bounding cube*.

Just like finding the closest five towns on a map will be faster on a state map than a world map, bounding reduces the points we need to look at.

We use cube\_enlarge(cube,radius,dimensions) to build an 18-dimensional cube that is some length (radius) wider than a point.

With our bounding hypercube, we can use a special cube operator, @>, which means *contains*.

This query finds the distance of all points contained within a five-unit cube of the *Star Wars* genre point.

**SELECT** title, cube\_distance(genre, *'(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'*) dist **FROM** movies **WHERE** cube\_enlarge(*'(0,7,0,0,0,0,0,0,0,7,0,0,0,0,10,0,0,0)'*::cube, 5, 18) @> genre **ORDER BY** dist;

Using a subselect, we can get the genre by movie name and perform our calculations against that genre using a table alias.

**SELECT** m.movie\_id, m.title **FROM** movies m, (**SELECT** genre, title **FROM** movies **WHERE** title = *'Mad Max'*) s **WHERE** cube\_enlarge(s.genre, 5, 18) @> m.genre **AND** s.title <> m.title **ORDER BY** cube\_distance(m.genre, s.genre) LIMIT 10;

.