PostgreSQL

Thursday 28 May 2015

09:28

PostgreSQL is a relational database management system

Despite the growing interest in newer database trends, the relational style remains the most popular and probably will for quite some time.

The prevalence of relational databases comes not only from their vast toolkits (triggers, stored procedures, advanced indexes), their data safety (via ACID compliance), or their mind share (many programmers speak and think relationally) but also from their query pliancy.

 If a relational schema is normalized, queries are flexible.

PostgreSQL is the finest open source example of the relational database management system (RDBMS) tradition.

That’s Post-greS-Q-L

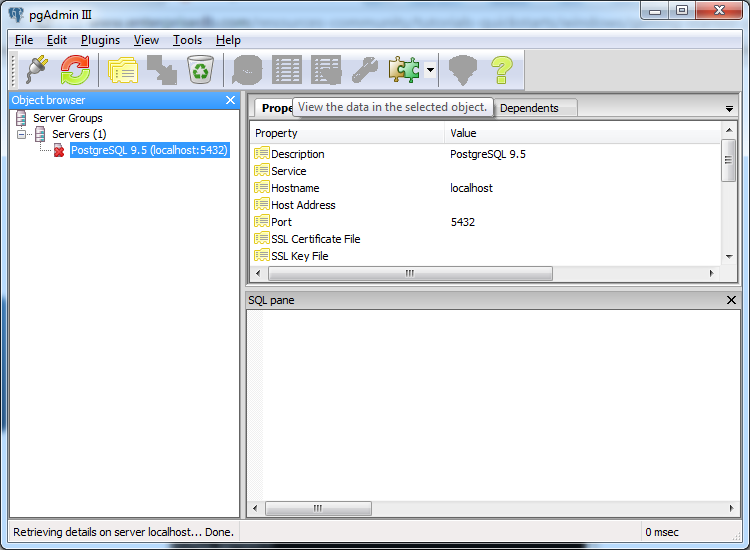
It has plug-ins for natural-language parsing, multidimensional indexing, geographic queries, custom datatypes, and much more.

It has sophisticated transaction handling, has built-in stored procedures for a dozen languages, and runs on a variety of platforms.

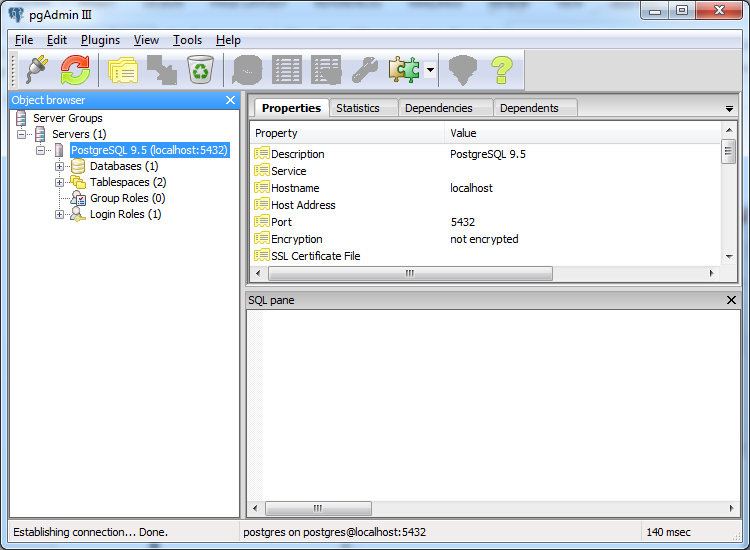
It’s fast and reliable, can handle terabytes of data, and has been proven to run in high-profile production projects such as Skype, France’s Caisse Nationale d’Allocations Familiales (CNAF), and the United States’ Federal Aviation Administration (FAA).

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

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; (Don’t forget the ;!!)**

When you get a connection to PostgreSQL it is always to a particular database. To access a different database, you must get a new connection. Using \c in psql closes the old connection and acquires a new one, using the specified database and/or credentials. You get a whole new back-end process and everything.

So run the command

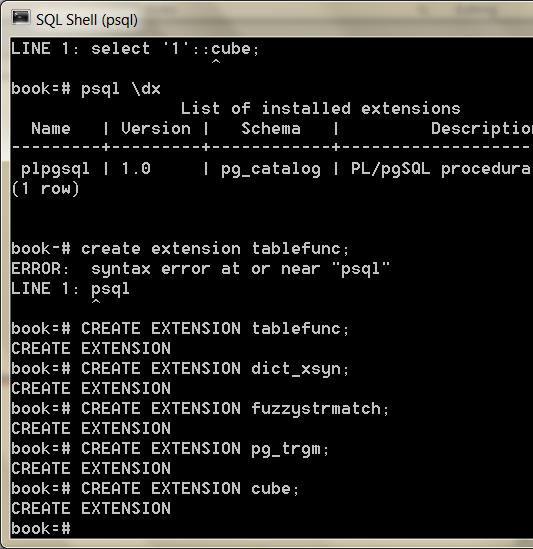
$ psql \c book or just \c book



Beyond the basic install, we’ll need to extend Postgres with the following contributed packages: tablefunc, dict\_xsyn, fuzzystrmatch, pg\_trgm, and cube.

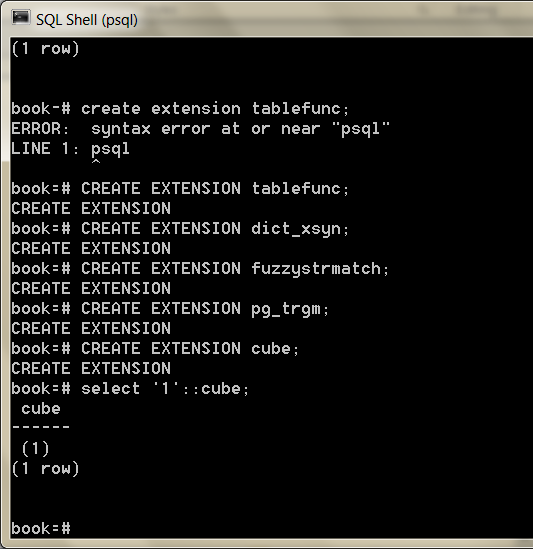
Try psql \dx to see installed extensions

Try CREATE EXTENSION tablefunc; if extension not there



Next, run the following command to ensure your contrib packages have been installed correctly:

**$ select '1'::cube;**



Seek out the online docs for more information if you receive an error message.

Relations, CRUD, and Joins

Like most databases Postgres provides a back-end server that does all of the work and a command-line shell to connect to the running server.

The server communicates through port 5432 by default, which you can connect to with the psql shell.

**$ psql \c book; (You can leave out the psql part of the command if you are in the shell)**

PostgreSQL prompts with the name of the database followed by a hash mark if you run as an administrator and by dollar sign as a regular user.

The shell also comes equipped with the best built-in documentation you will find in any console.

Typing \h lists information about SQL commands, and \? helps with psql-specific commands, namely, those that begin with a backslash.

You can find usage details about each SQL command in the following way:

*\h CREATE INDEX*

Command: **CREATE INDEX**

Description: define a new **index**

Syntax:

**CREATE** [ **UNIQUE** ] **INDEX** [ CONCURRENTLY ] [ name ] **ON table** [ **USING** method ]

( { column | ( expression ) } [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | ... [ WITH ( storage\_parameter = value [, ... ] ) ]

[ TABLESPACE tablespace ]

[ **WHERE** predicate ]

It’s worth looking over (or brushing up on) a few common commands, like SELECT or CREATE TABLE.

Starting with SQL

PostgreSQL follows the SQL convention of calling relations TABLEs, attributes COLUMNs, and tuples ROWs.

Working with Tables

PostgreSQL, being of the relational style, is a design-first datastore.

First you design the schema, and then you enter data that conforms to the definition of that schema.

Creating a table consists of giving it a name and a list of columns with types and (optional) constraint information.

Each table should also nominate a unique identifier column to pinpoint specific rows.

That identifier is called a PRIMARY KEY. The SQL to create a countries table looks like this:

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

This new table will store a set of rows, where each is identified by a two- character code and the name is unique.

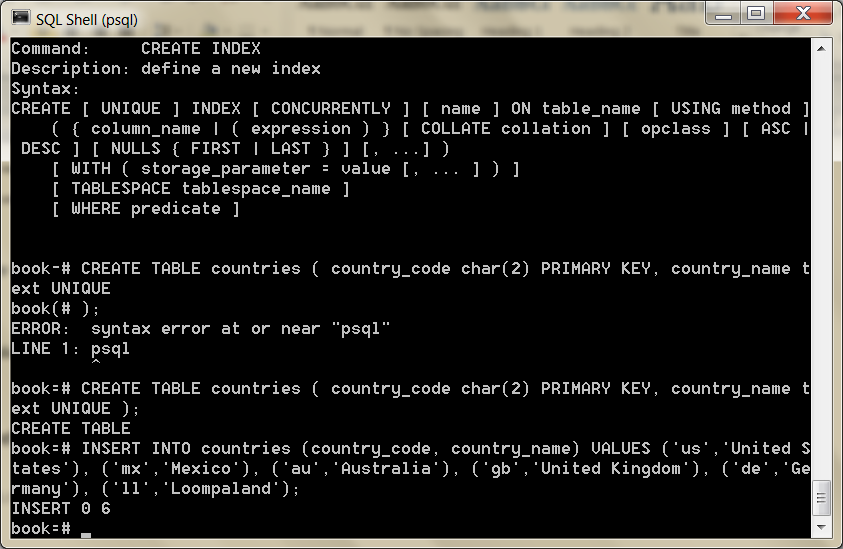
These columns both have *constraints*.

The PRIMARY KEY constrains the country\_code column to disallow duplicate country codes.

 We explicitly gave country\_name a similar unique constraint, although it is not a primary key.

We can populate the countries table by inserting a few rows.

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



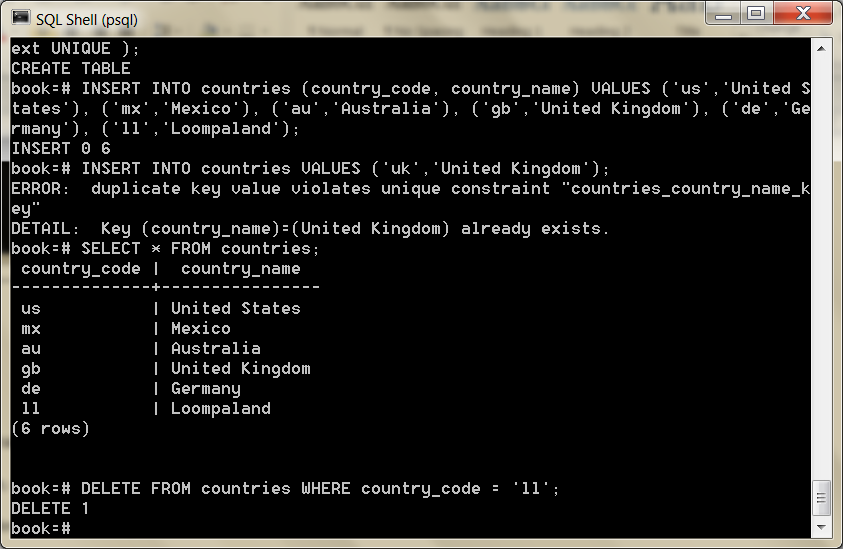
We can validate that the proper rows were inserted by reading them using the SELECT...FROM table command.

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



According to any respectable map, Loompaland isn’t a real place—let’s remove it from the table.

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

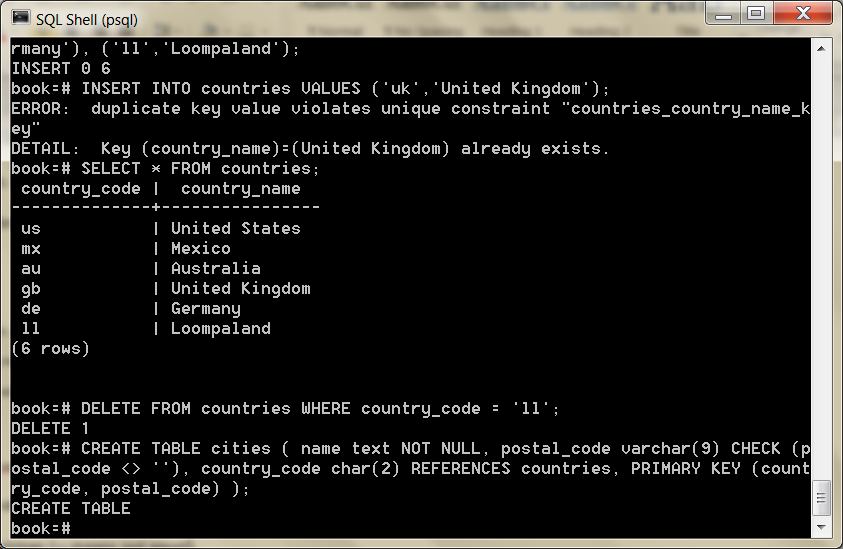


With only real countries left in the countries table, let’s add a cities table.

To ensure any inserted country\_code also exists in our countries table, we add the REFERENCES keyword.

Since the country\_code column references another table’s key, it’s known as the *foreign key* constraint.

**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) );



This time, we constrained the name in cities by disallowing NULL values.

We constrained postal\_code by checking that no values are empty strings (<> means *not equal*).

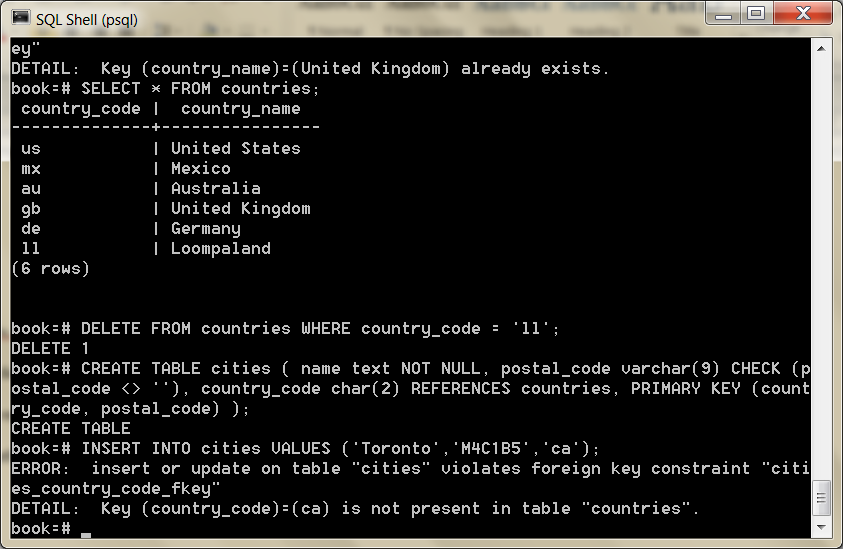
Furthermore, since a PRIMARY KEY uniquely identifies a row, we created a compound key: country\_code + postal\_code.

 Postgres also has a rich set of datatypes.

You’ve just seen three different string representations: text (a string of any length), varchar(9) (a string of variable length up to nine characters), and char(2) (a string of exactly two characters).

With our schema in place, let’s insert *Toronto, CA*.

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



This failure is good! Since country\_code REFERENCES countries, the country\_code must exist in the countries table.

This is called *maintaining referential integrity*, and ensures our data is always correct.

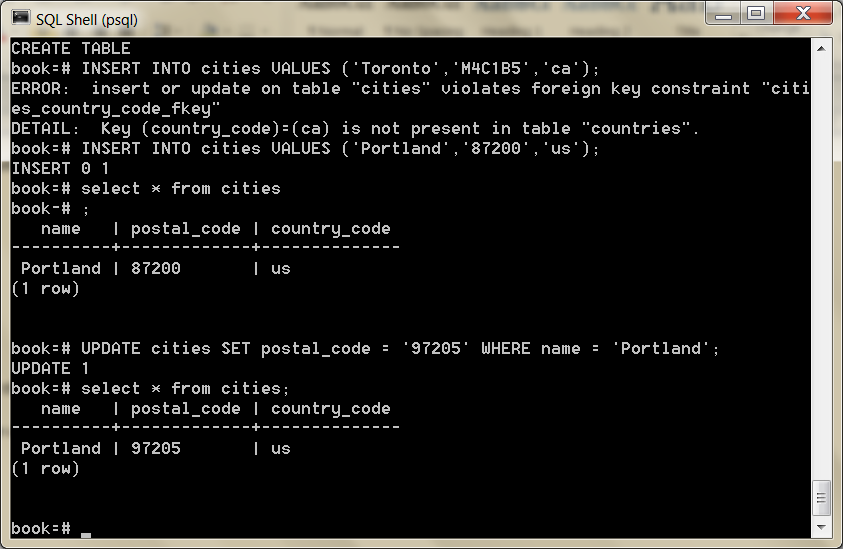
Now let’s try another insert, this time with a U.S. city.

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

This is a successful insert, to be sure. But we mistakenly entered the wrong postal\_code.

The correct postal code for Portland is *97205*. Rather than delete and reinsert the value, we can update it inline.

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



We have now Created, Read, Updated, and Deleted table rows.

Join Reads

What sets relational databases like PostgreSQL apart is their ability to join tables together when reading them.

Joining, in essence, is an operation taking two separate tables and combining them in some way to return a single table.

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;



The join returns a single table, sharing all columns’ values of the cities table plus the matching country\_name value from the countries table.

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.

A venue exists in both a *postal code* and a specific *country*.

The *foreign key* must be two columns that reference both cities *primary key* columns.

(MATCH FULL is a constraint that ensures either both values exist or both are NULL.)

**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);

This venue\_id column is a common primary key setup: automatically incremented integers (1, 2, 3, 4, and so on...).

We make this identifier using the SERIAL keyword (MySQL has a similar construct called AUTO\_INCREMENT).

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



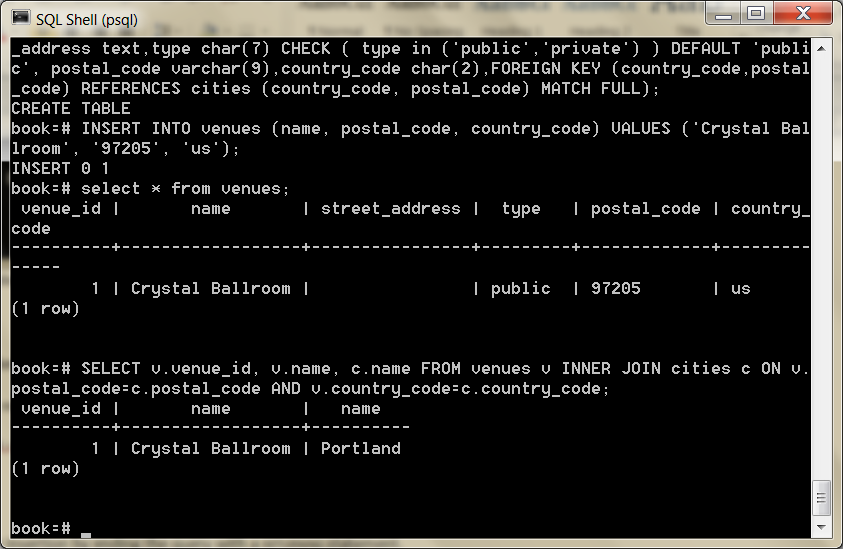
Although we did not set a venue\_id value, creating the row populated it.

Back to our compound join.

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;



This provides the new venue\_id without issuing another query.

Outer Joins

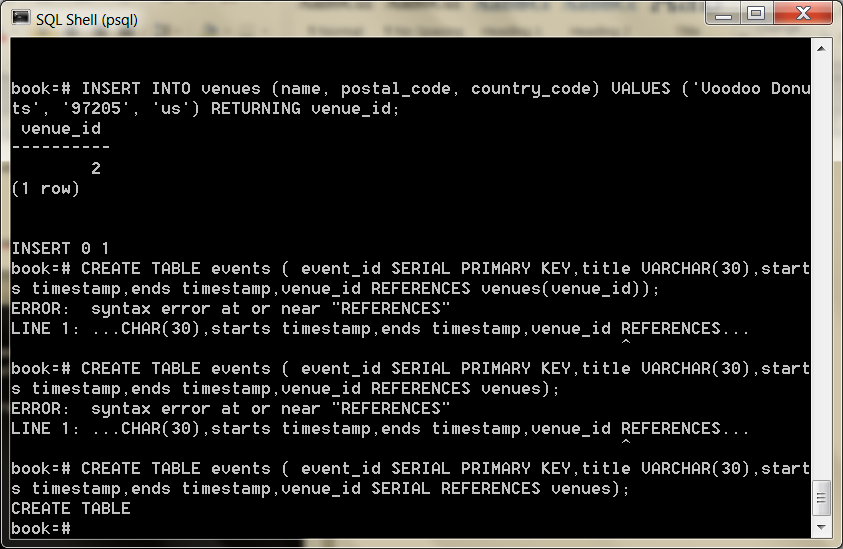
In addition to inner joins, PostgreSQL can also perform *outer joins*.

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

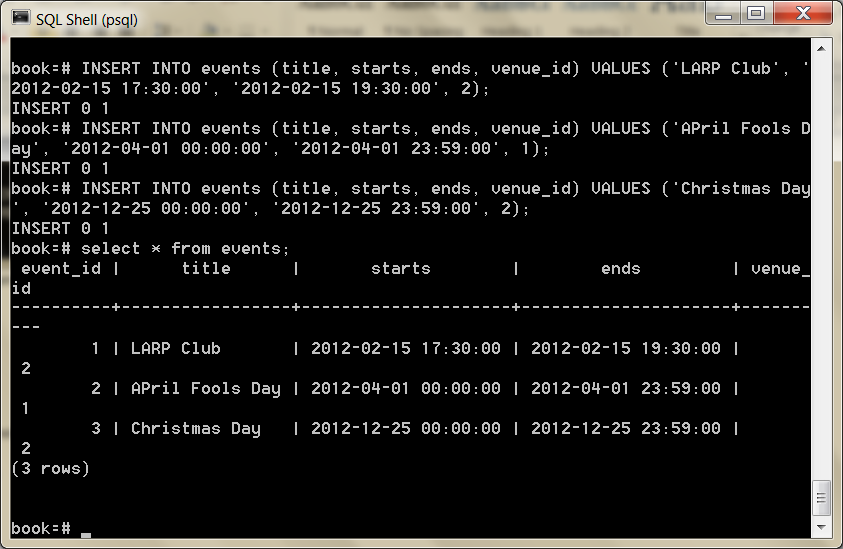
Use: **INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (*'LARP Club'*, *'2012-02-15 17:30'*, *'2012-02-15 19:30', 2);*

title | starts | ends | venue\_id | event\_id

*----------------+---------------------+---------------------+----------+---------*

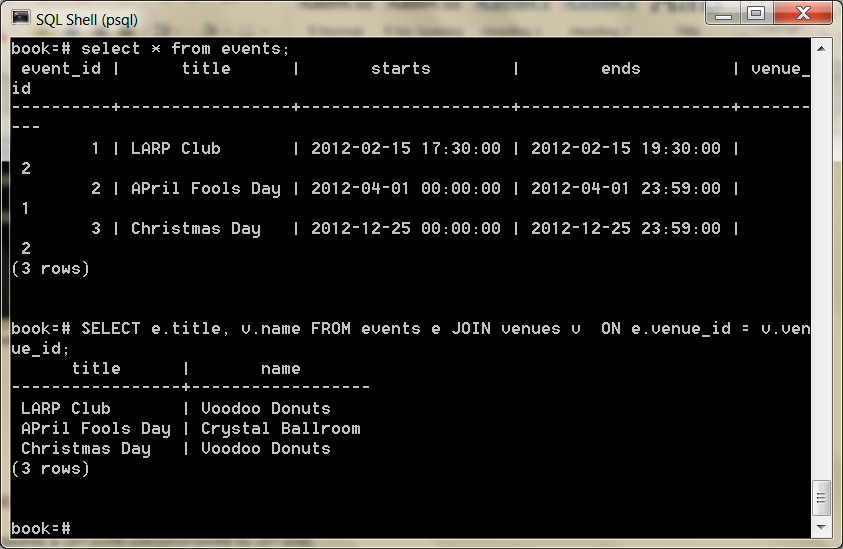
LARP Club | 2012-02-15 17:30:00 | 2012-02-15 19:30:00 |  
April Fools Day | 2012-04-01 00:00:00 | 2012-04-01 23:59:00 |  
Christmas Day | 2012-12-25 00:00:00 | 2012-12-25 23:59:00 |

2 | 1 | 2 | 3



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).

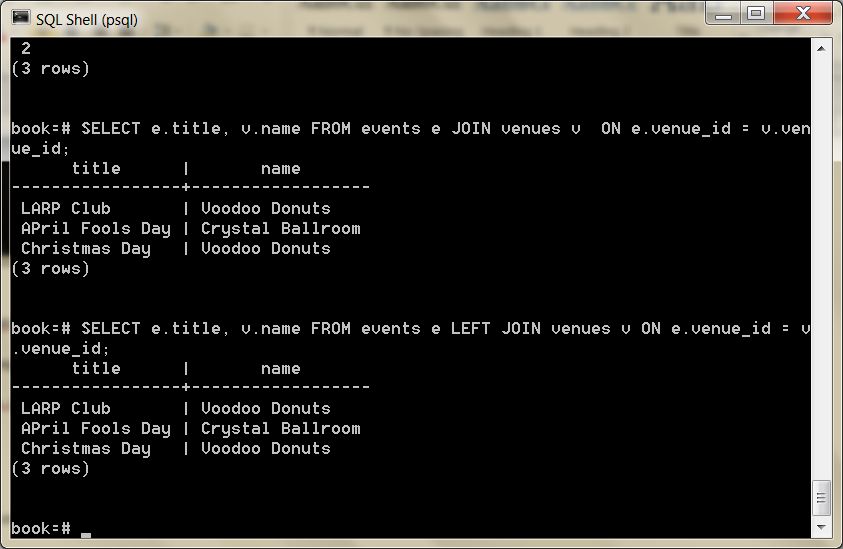
**SELECT** e.title, v.name **FROM** events e **JOIN** venues v **ON** e.venue\_id = v.venue\_id;



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

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

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

Fast Lookups with Indexing

The speed of PostgreSQL (and any other RDBMS) lies in its efficient management of blocks of data, reducing disk reads, query optimization, and other techniques.

But those go only so far in fetching results fast.

If we select the title of *Christmas Day* from the events table, the algorithm must scan every row for a match to return.

Without an *index*, each row must be read from disk to know whether a query should return it.

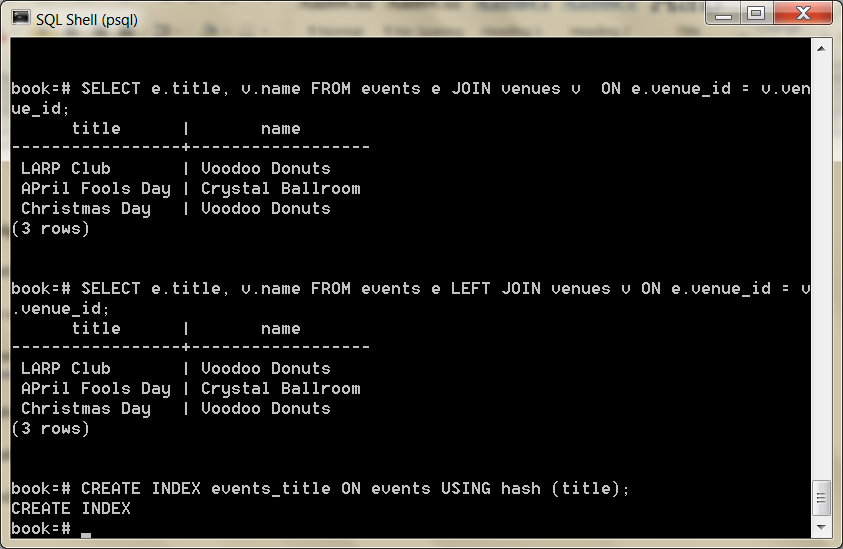
An index is a special data structure built to avoid a full table scan when performing a query.

PostgreSQL automatically creates an index on the primary key, where the key is the primary key value and where the value points to a row on disk

Using the UNIQUE keyword is another way to force an index on a table column.

You can explicitly add a hash index using the CREATE INDEX command, where each value must be unique (like a hashtable or a map).

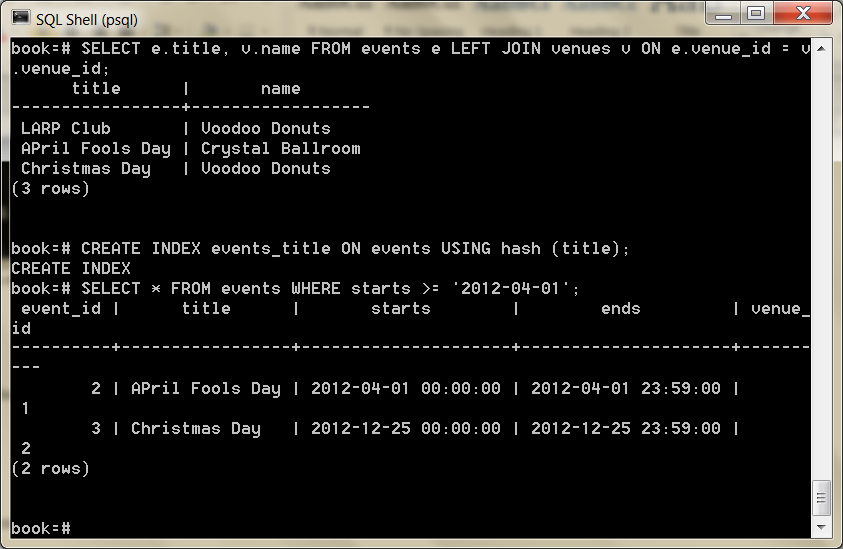
**CREATE INDEX** events\_title **ON** events **USING** hash (title);



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 >= *'2012-04-01'*;



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

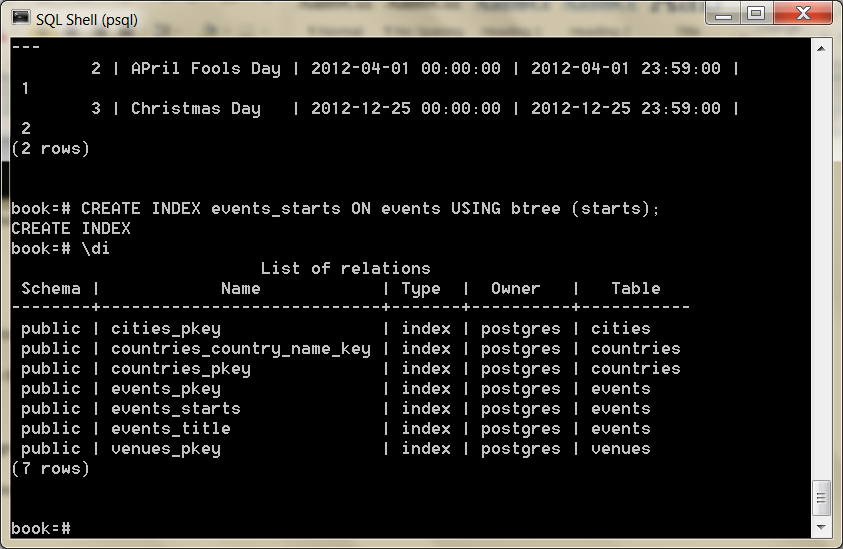
**CREATE INDEX** events\_starts **ON** events **USING** btree (starts);

Now our query over a range of dates will avoid a full table scan.

It makes a huge difference when scanning millions or billions of rows.

We can inspect our work with this command to list all indexes in the schema:

*\di*



Stored Procedures

Every command we’ve seen until now has been declarative, but sometimes we need to run some code.

At this point, you must make a decision: execute code on the client side or execute code on the database side.

Stored procedures can offer huge performance advantages for huge architectural costs. You may avoid streaming thousands of rows to a client application, but you have also bound your application code to this database.

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.

postgres/add\_event.sql

**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;

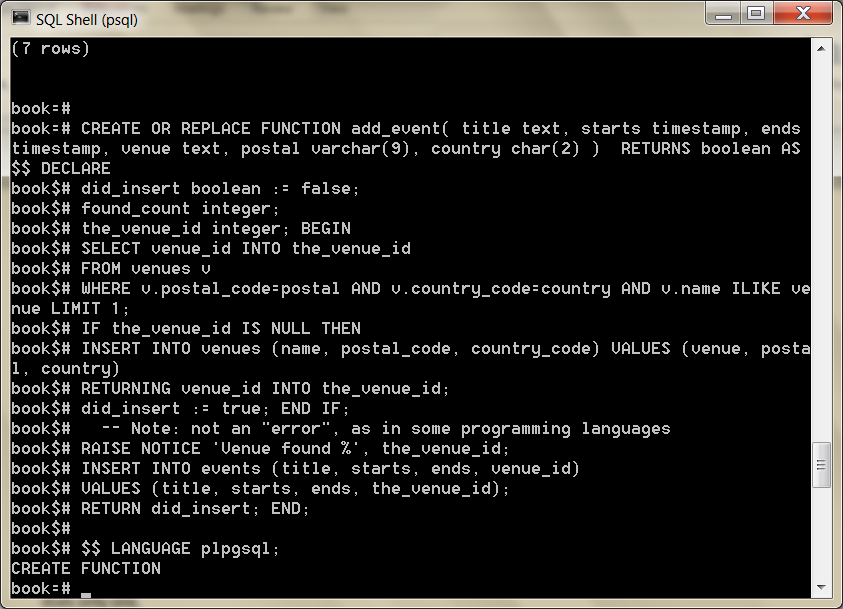
**INSERT INTO** events (title, starts, ends, venue\_id) **VALUES** (title, starts, ends, the\_venue\_id);

RETURN did\_insert;

**END**;

$$ LANGUAGE plpgsql;

Try to copy and paste the code above



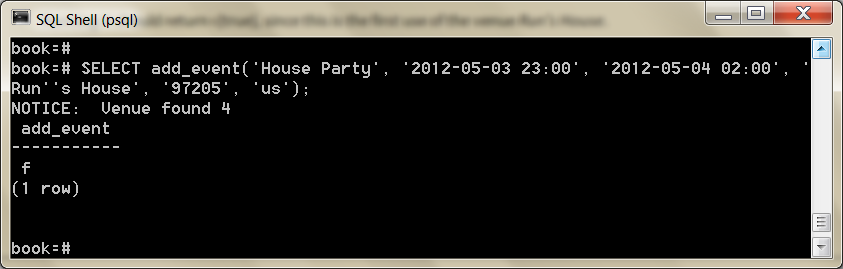
You can import this external file into the current schema by the following command-line argument (if you don’t feel like typing all that code).

book=*# \i add\_event.sql*

Running it 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.

**SELECT** add\_event(*'House Party'*, *'2012-05-03 23:00'*, *'2012-05-04 02:00'*, *'Run''s House'*, *'97205'*, *'us'*);



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

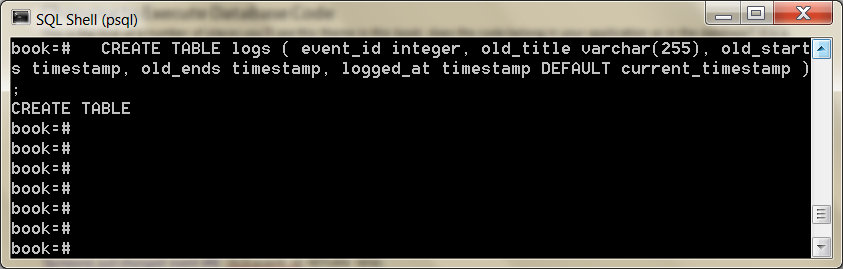
Triggers automatically fire stored procedures when some event happens, like an insert or update.

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.

Choosing to Execute Database Code

*does the code belong in your application or in the database? It is a difficult decision—one that you’ll have to answer uniquely for every application.*

*The benefit is you’ll often improve performance by as much as an order of magnitude. For example, you might have a complex application-specific calculation that requires custom code. If the calculation involves many rows, a stored procedure will save you from moving thousands of rows instead of a single result. The cost is splitting your application, your code, and your tests, across two different programming paradigms*.

postgres/log\_event.sql

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

$$ DECLARE

**BEGIN**

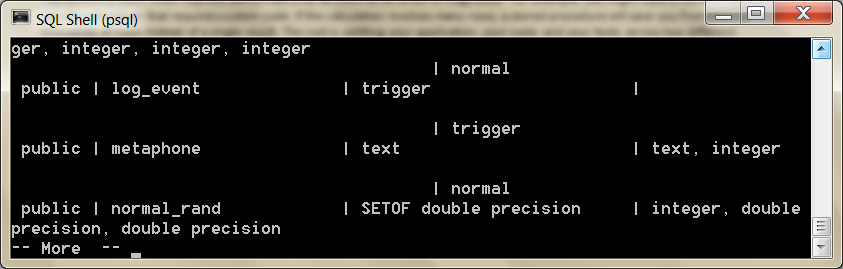
**INSERT INTO** logs (event\_id, old\_title, old\_starts, old\_ends) **VALUES** (OLD.event\_id, OLD.title, OLD.starts, OLD.ends);

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

**END**;

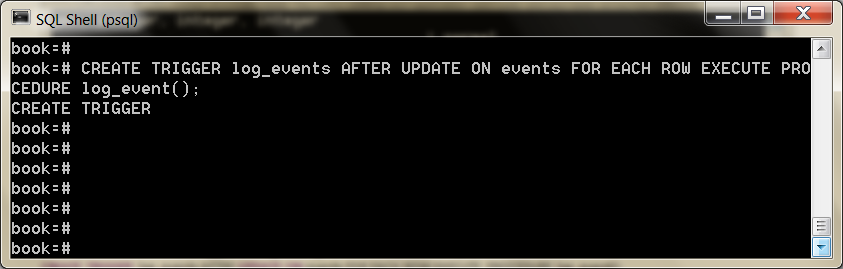
$$ LANGUAGE plpgsql;

\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();

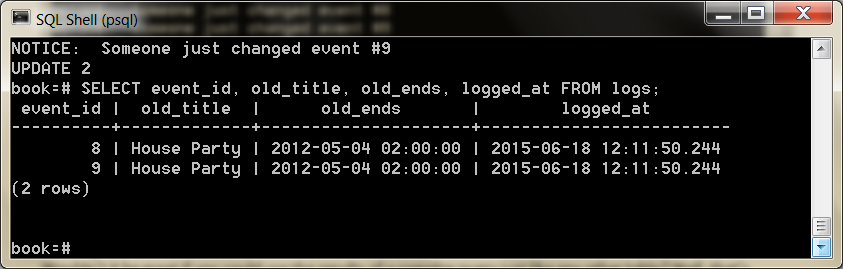


So, it turns out our party at Run’s House has to end earlier than we hoped. Let’s change the event.

**UPDATE** events **SET** ends=*'2012-05-04 01:00:00'* **WHERE** title=*'House Party'*;



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



Triggers can also be created before updates and before or after inserts.

**Dump and Restore**

The idea behind this dump method is to generate a text file with SQL commands that, when fed back to the server, will recreate the database in the same state as it was at the time of the dump.

PostgreSQL provides the utility program **[pg\_dump](https://www.postgresql.org/docs/9.1/app-pgdump.html)** for this purpose. The basic usage of this command is:

pg\_dump ***dbname*** > ***outfile***

The text files created by pg\_dump are intended to be read in by the psql program. The general command form to restore a dump is

psql -Upostgres -dbook -1 -f pgbackup.sql

See: <https://www.postgresql.org/docs/9.1/backup-dump.html>

The database ***dbname*** will not be created by this command, so you must create it yourself from template0 before executing psql (e.g., with createdb -T template0 ***dbname***)

Viewing the World

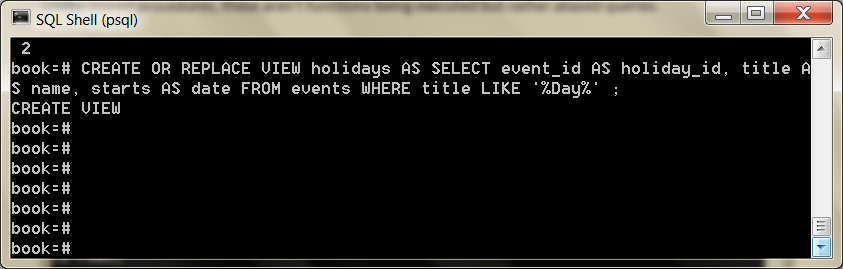
Wouldn’t it be great if you could use the results of a complex query just like any other table? Well, that’s exactly what VIEWs are for

Unlike stored procedures, these aren’t functions being executed but rather aliased queries.

In our database, all holidays contain the word *Day* and have no venue.

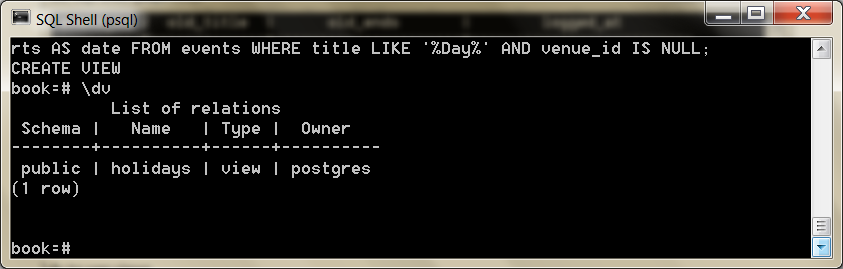
postgres/holiday\_view\_1.sql

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



So, creating a view is as simple as writing a query and prefixing it with CREATE VIEW some\_view\_name AS.

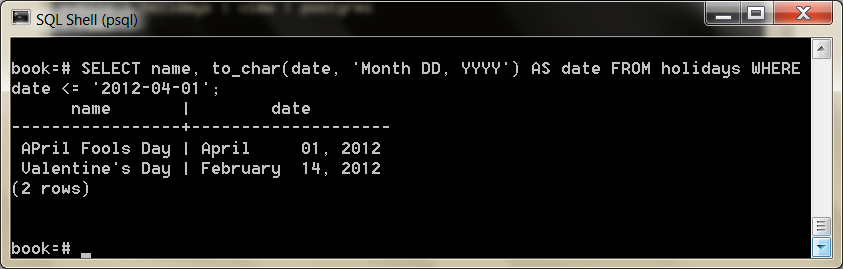
\dv to see views



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 *2012- 02-14* to events and query the holidays view.

**SELECT** name, to\_char(**date**, *'Month DD, YYYY'*) **AS date FROM** holidays **WHERE date** <= *'2012-04-01'*;



Views are powerful tools for opening up complex queried data in a simple way.

The query may be a roiling sea of complexity underneath, but all you see is a table.

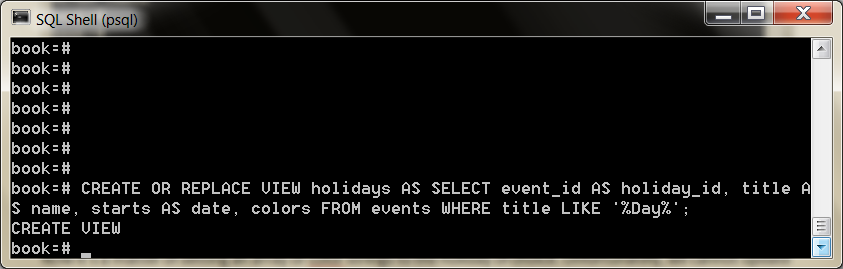
If you want to add a new column to the view, it will have to come from the underlying table. Let’s alter the events table to have an array of associated colors.

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. Unfortunately, we cannot update a view directly.

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

ERROR: cannot update a view

HINT: You need an unconditional ON UPDATE DO INSTEAD rule.

Looks like we need a RULE.

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

Every time Postgres runs an SQL statement, it parses the statement into a query tree (generally called an *abstract syntax tree*).

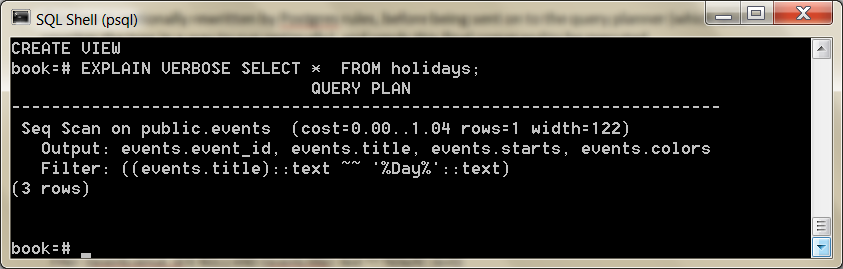
Operators and values become branches and leaves in the tree, and the tree is walked, pruned, and in other ways edited before execution.

This tree is optionally rewritten by Postgres rules, before being sent on to the query planner (which also rewrites the tree in a way to run optimally), and sends this final command to be executed.

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

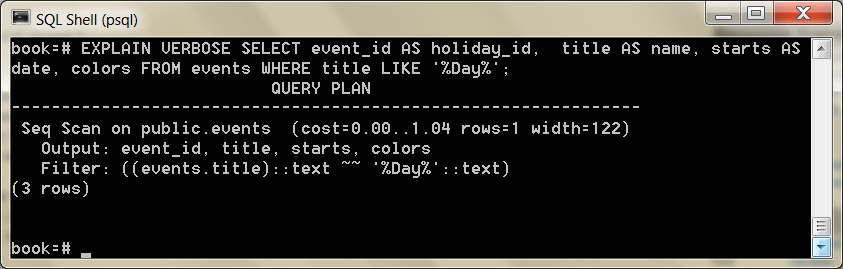
We can prove this by taking a look at the execution plan of the holidays view using the EXPLAIN command (notice *Filter* is the WHERE clause, and *Output* is the column list).

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%'* ;



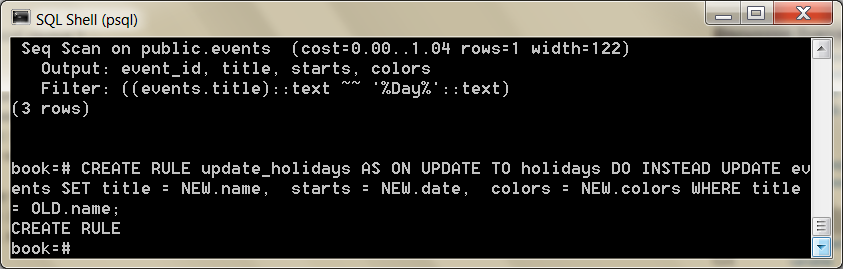
So, to allow updates against our holidays view, we need to craft a RULE that tells Postgres what to do with an UPDATE.

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.

postgres/create\_rule.sql

**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;

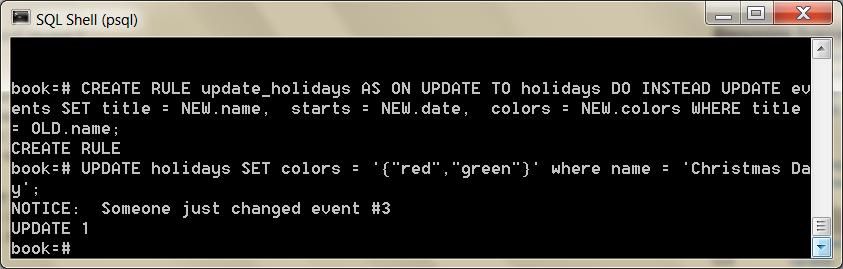


With this rule in place, now we can update holidays directly.

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

Next let’s insert *New Years Day* on *2013-01-01* 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.

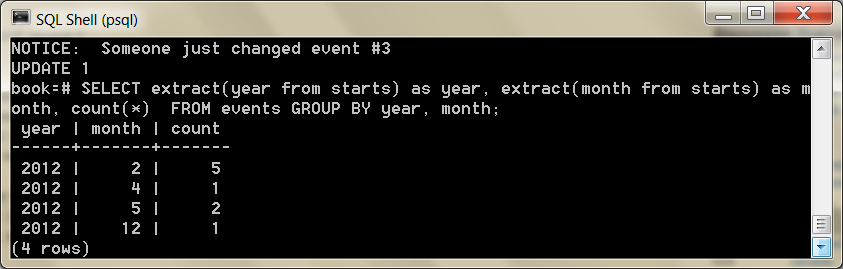
This kind of operation is commonly done by a *pivot table*.

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).

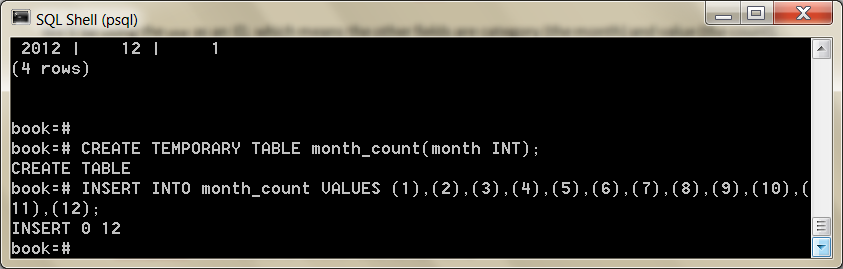
The crosstab() function needs another set of values to represent months.

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.

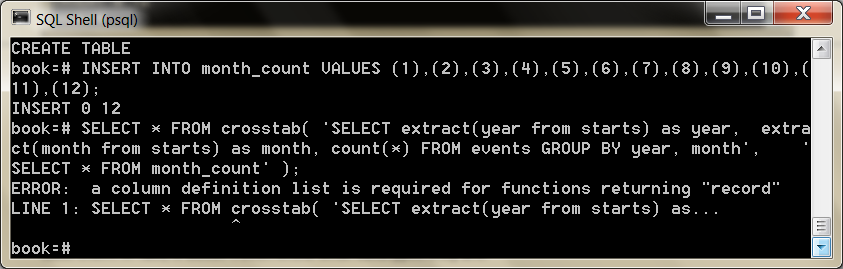
**CREATE** TEMPORARY **TABLE** month\_count(month **INT**);

**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.

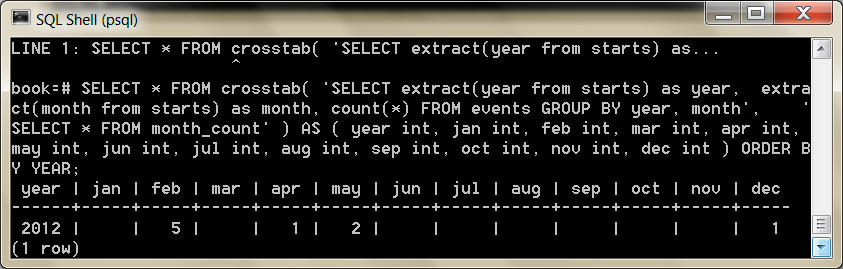
It may feel cryptic, but it’s saying the function is returning a set of records (rows), but it doesn’t know how to label them. In fact, it doesn’t even know what datatypes they are.

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.



Go ahead and add a couple more events on another year just to see next year’s event counts. Run the crosstab function again, and enjoy the calendar.

 Full-Text and Multidimensions

We’ll investigating the many tools at our disposal to build a movie query system.

We’ll begin with the many ways that PostgreSQL can search actor/movie names using fuzzy string matching.

Then we’ll discover the cube package by creating a movie suggestion system based on similar genres of movies we already like.

Since these are all contributed packages, the implementations are special to PostgreSQL and not part of the SQL standard.

Commonly, when designing a relational database schema, you’ll start with an entity diagram.

We’ll be writing a personal movie suggestion system that keeps track of movies, their genres, and their actors.

As a reminder we installed several contributed packages.

Again, the list we’ll need installed is as follows: tablefunc, dict\_xsyn, fuzzystrmatch, pg\_trgm, and cube.

Let’s first build the database.

It’s often good practice to create indexes on foreign keys to speed up reverse lookups (such as what movies this actor is involved in).

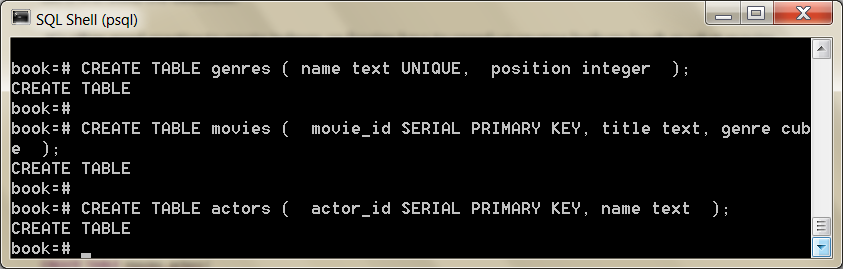
You should also set a UNIQUE constraint on join tables like movies\_actors to avoid duplicate join values.

postgres/create\_movies.sql

**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**  );

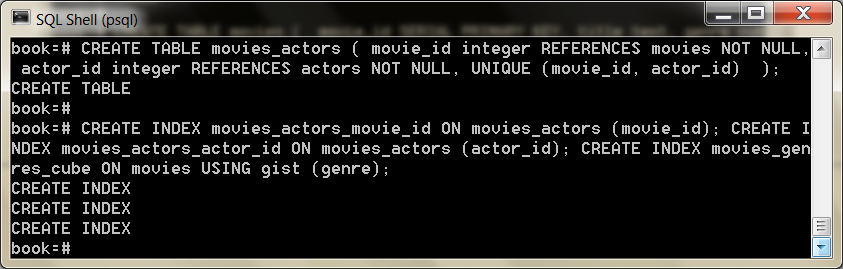


**CREATE TABLE** movies\_actors ( movie\_id **integer** REFERENCES movies **NOT** NULL, actor\_id **integer** REFERENCES actors **NOT** NULL, **UNIQUE** (movie\_id, actor\_id) );

**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);

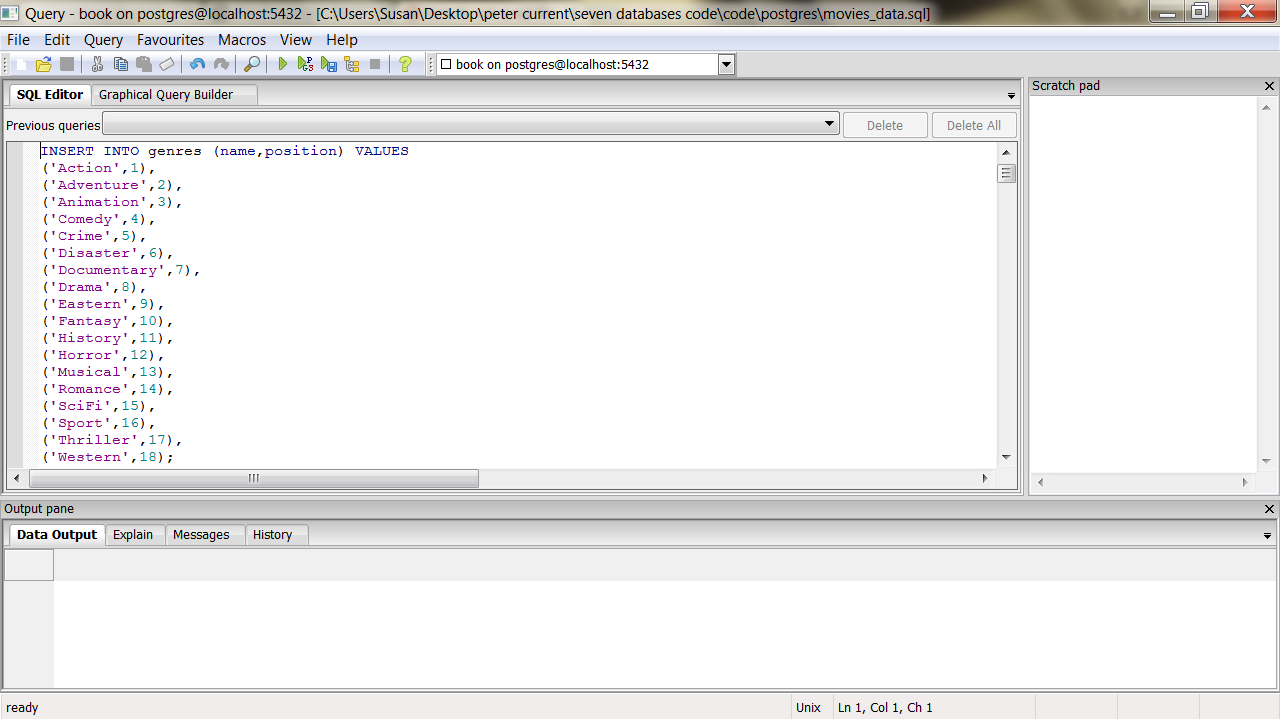


You can download the movies\_data.sql file as a file and populate the tables by piping the file into the database.

I started pgAdmin and connected to the book database

I selected the query tool from tools

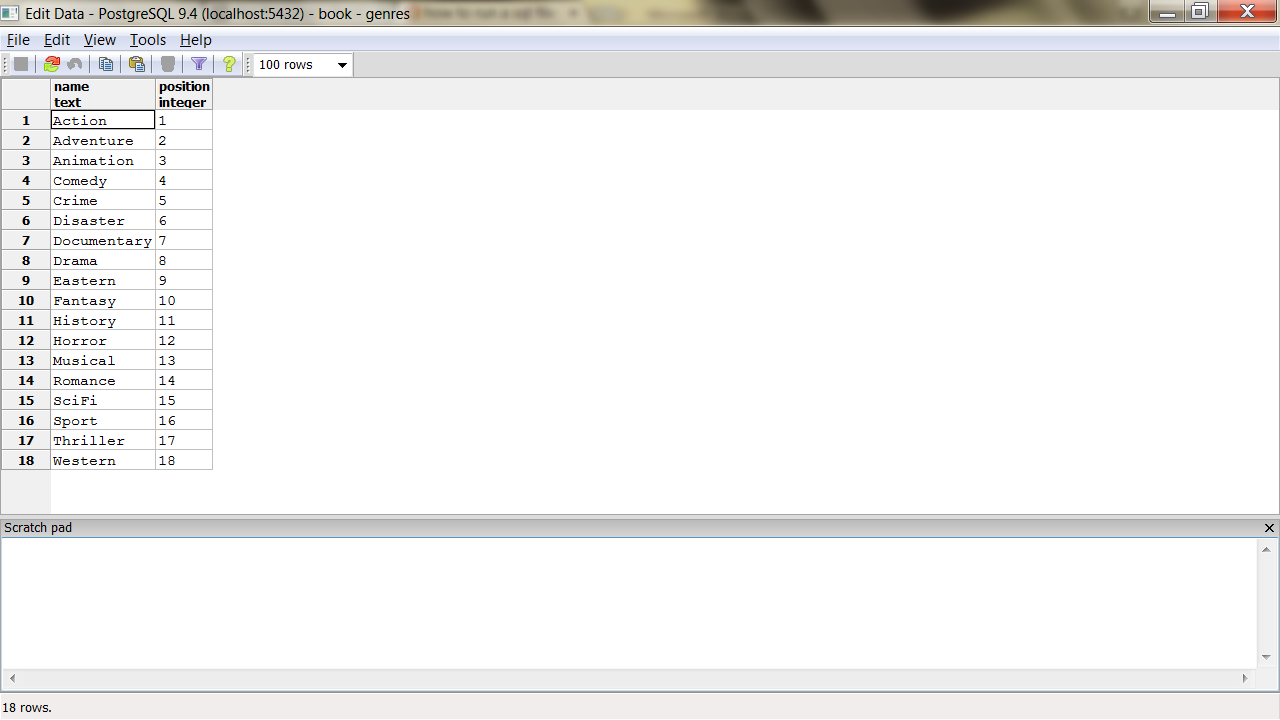
I opened the movies\_data.sql file



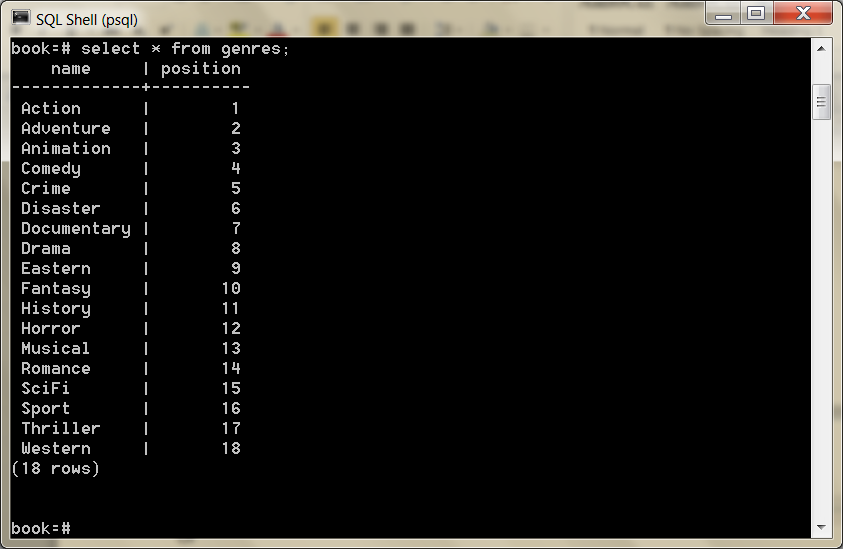
I ran the query using Execute pgScript (or Execute?)

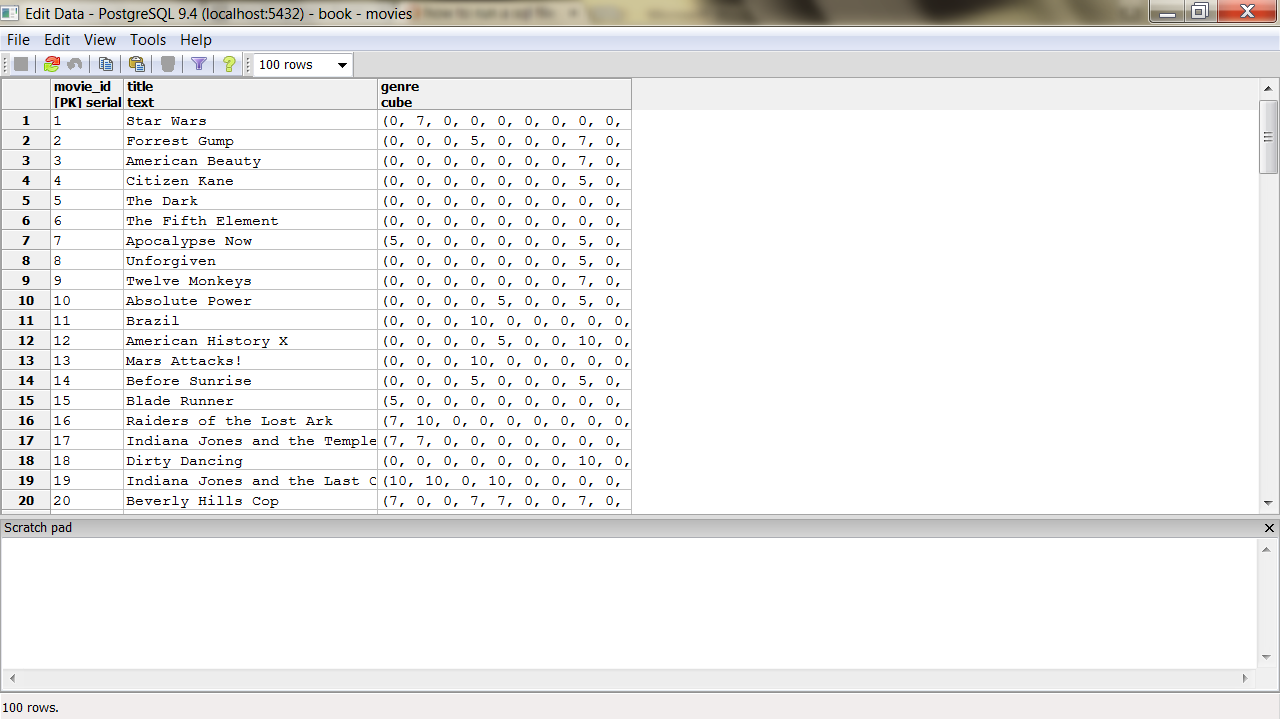
Or try method from command line above

Now I could see the data in pgAdmin:



Or





Fuzzy Searching

Opening up a system to text searches means opening your system to inaccurate inputs.

You have to expect typos like “Brid of Frankstein.”

Sometimes, users can’t remember the full name of “J. Roberts.”

In other cases, we just plain don’t know how to spell “Benn Aflek.”

We’ll look into a few PostgreSQL packages that make text searching easy.

It’s worth noting that as we progress, this kind of string matching blurs the lines between relational queries and searching frameworks like Lucene.

Although some may feel features like full- text search belong with the application code, there can be performance and administrative benefits of pushing these packages to the database, where the data lives.

SQL Standard String Matches

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

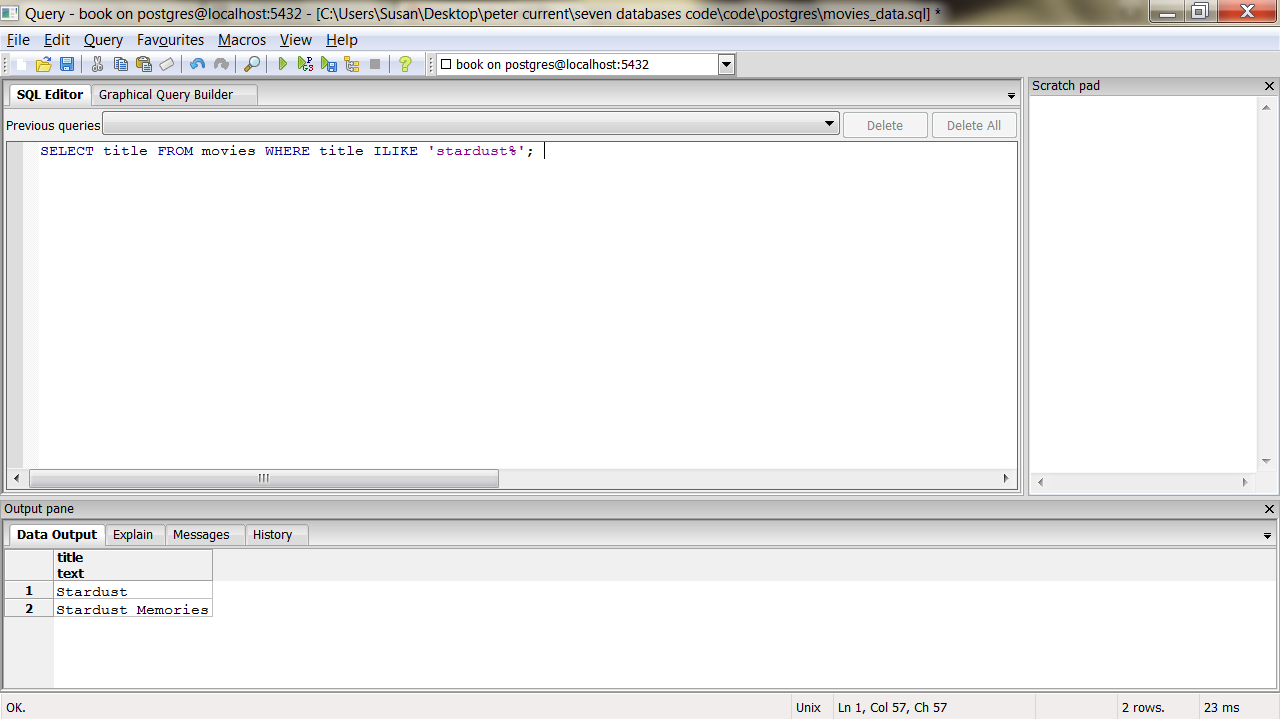
LIKE and ILIKE

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

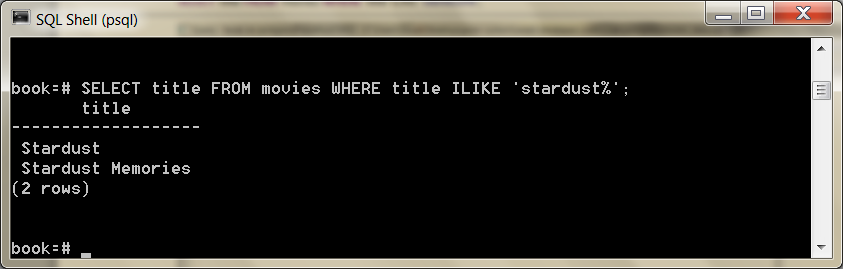
They are fairly universal in relational databases. LIKE compares column values against a given pattern string.

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

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

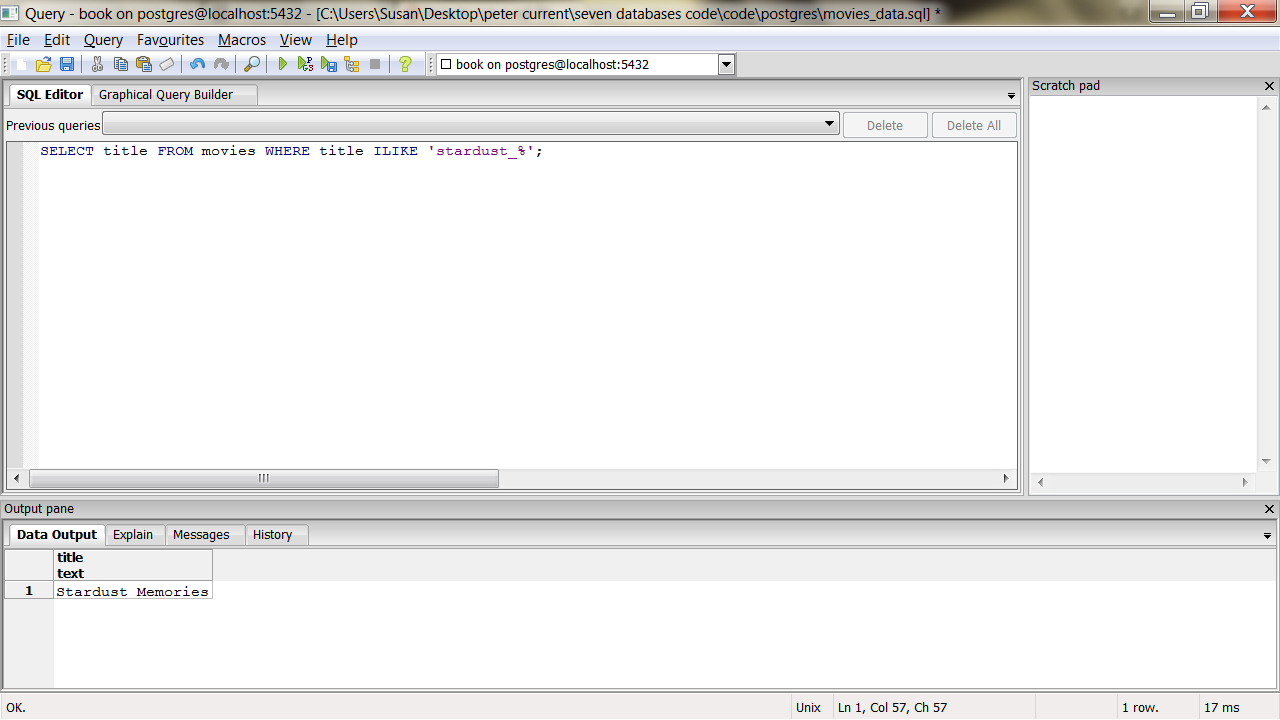


Or

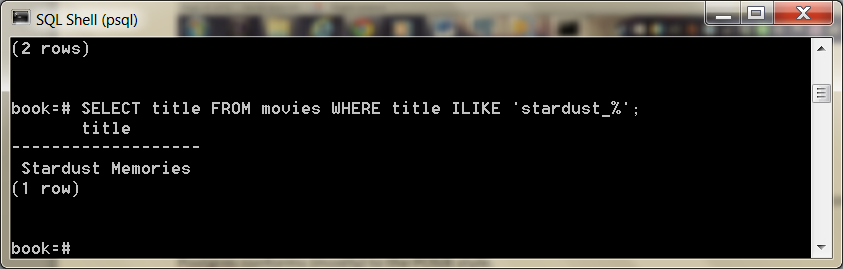


If we want to be sure the substring *stardust* is not at the end of the string, we can use the underscore (\_) character as a little trick.

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



Or



This is useful in basic cases, but LIKE is limited to simple wildcards.

Regex

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

Many databases support them.

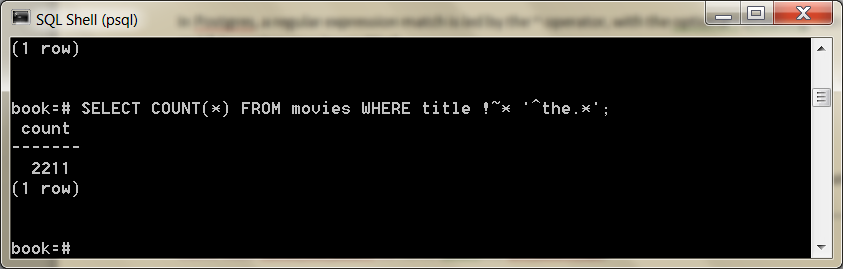
There are entire books dedicated to writing powerful expressions

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*).

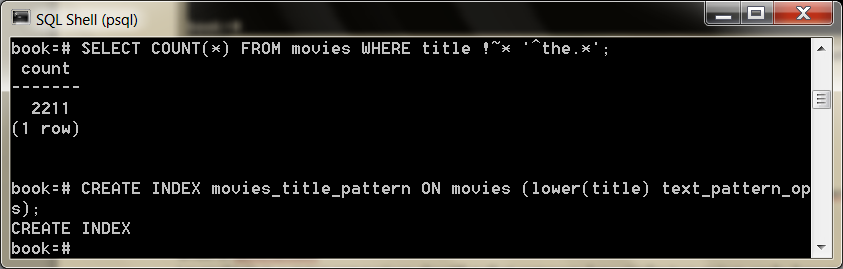
So, to count all movies that do *not* begin with *the*, the following case-insensitive query will work. The characters inside the string are the regular expression.

**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);



We used the text\_pattern\_ops because the title is of type text.

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

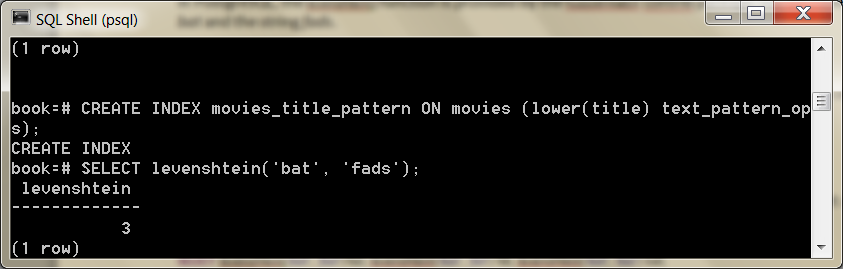
Levenshtein is a string comparison algorithm that compares how similar two strings are by how many *steps* are required to change one string into another.

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'*);

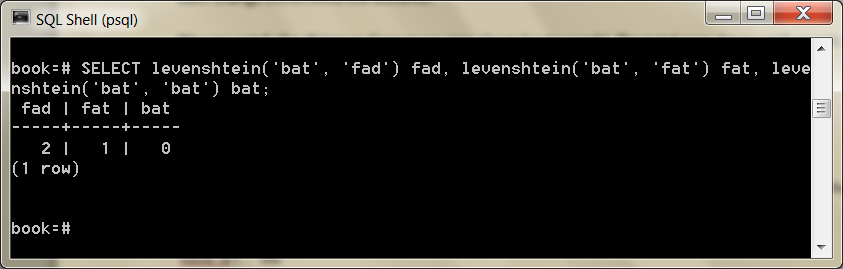


The Levenshtein distance is 3 because—compared to the string *bat*—we replaced two letters (b=>f, t=>d), and we added a letter (+s).

Each change increments the distance.

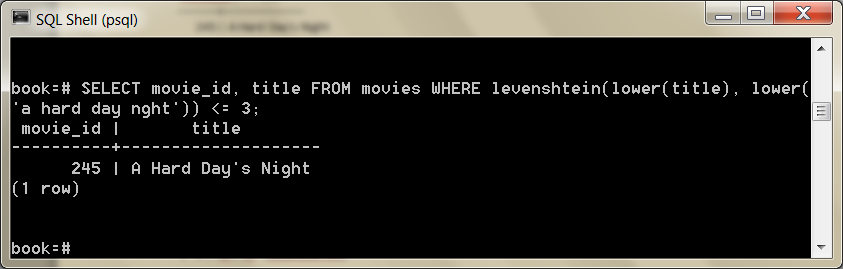
We can watch the distance close as we step closer (so to speak). The total goes down until we get zero (the two strings are equal).

**SELECT** levenshtein(*'bat'*, *'fad'*) fad, levenshtein(*'bat'*, *'fat'*) fat, levenshtein(*'bat'*, *'bat'*) bat;



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;



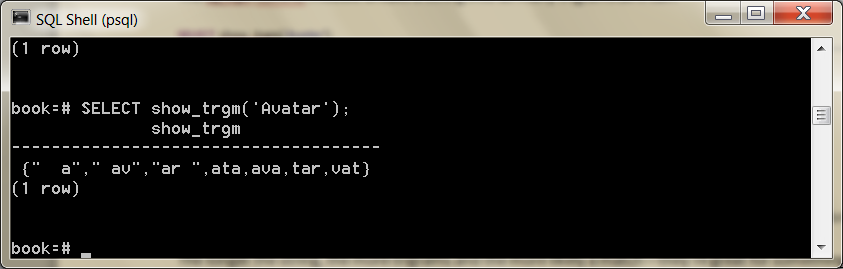
This ensures minor differences won’t over-inflate the distance.

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.

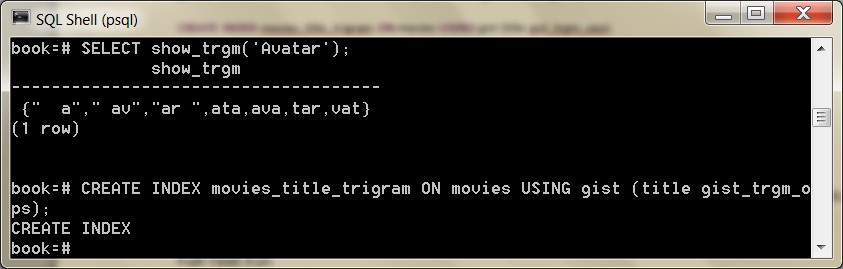
The strings with the most matches are the most similar.

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.

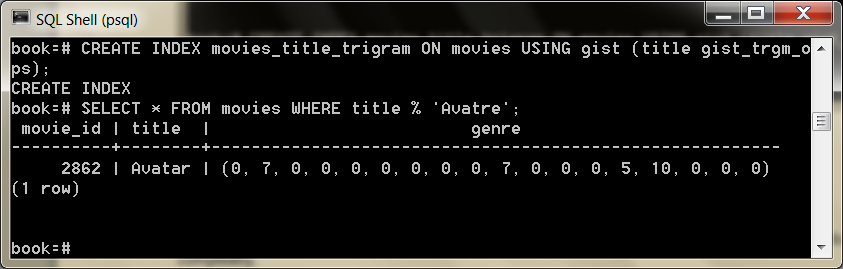
We’ll create a trigram index against movie names to start (we use Generalized Index Search Tree [GIST], a generic index API made available by the PostgreSQL engine).

**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'*;



Trigrams are an excellent choice for accepting user input, without weighing them down with wildcard complexity.

* HERE 19/11

Full-Text Search

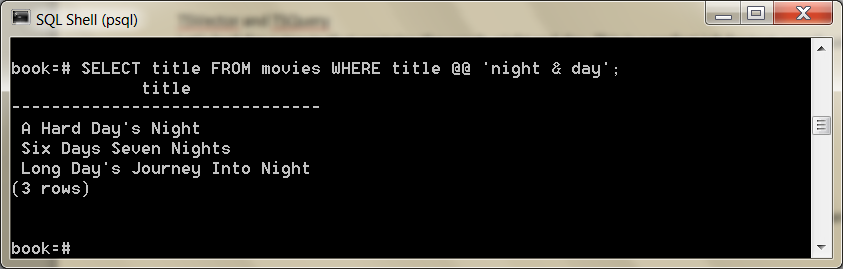
Next, we want to allow users to perform full-text searches based on matching words, even if they’re pluralized.

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.

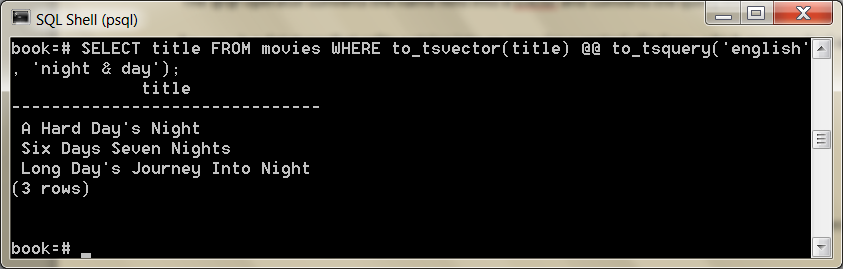
The @@ operator converts the name field into a tsvector and converts the query into a tsquery.

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.

The language corresponds to a dictionary (which we’ll see more of in a few paragraphs).

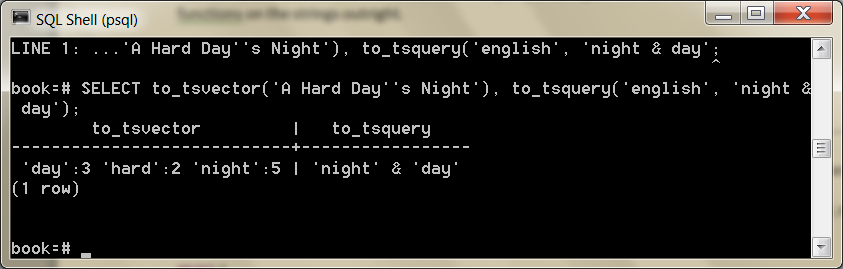
The previous query is equivalent to the following (if your system language is set to English):

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



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.

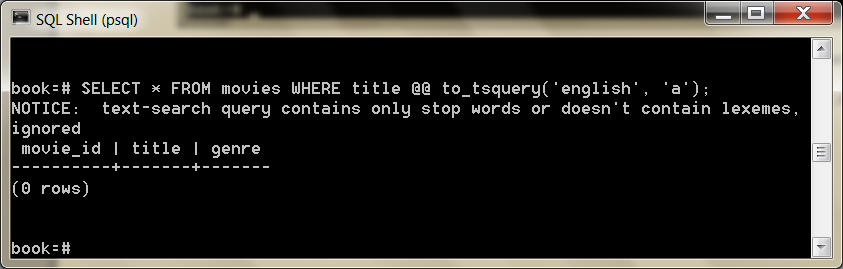
**SELECT** to\_tsvector(*'A Hard Day''s Night'*), to\_tsquery(*'english'*, *'night & day'*);



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.

The English dictionary was used by the parser to normalize our string into useful English components.

In your console, you can view the output of the stop words under the English tsearch\_data directory.

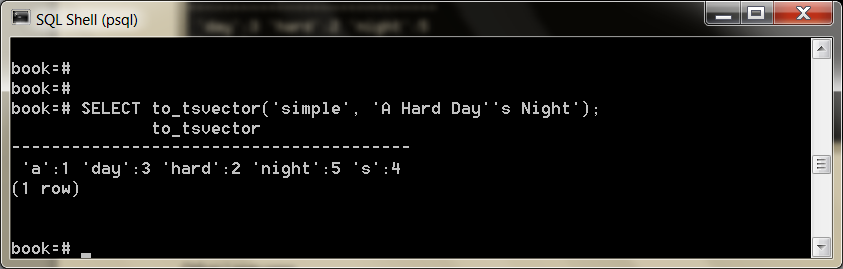
cat `pg\_config --sharedir`/tsearch\_data/english.stop (not working on windows)

We could remove *a* from the list, or we could use another dictionary like simple that just breaks up strings by nonword characters and makes them lowercase.

Compare these two vectors:

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

**SELECT** to\_tsvector(*'simple'*, *'A Hard Day''s Night'*);

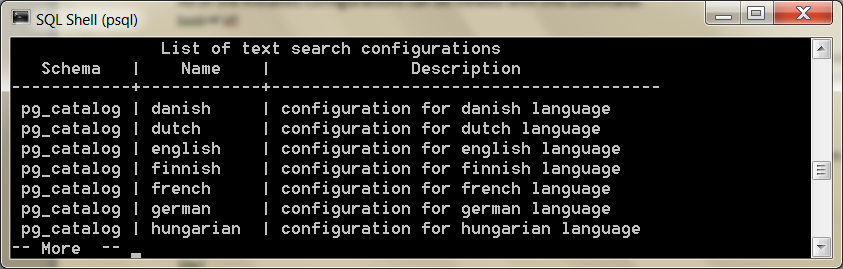
With simple, you can retrieve any movie containing the lexeme *a*.

Other Languages

Since Postgres is doing some natural-language processing here, it only makes sense that different configurations would be used for different languages.

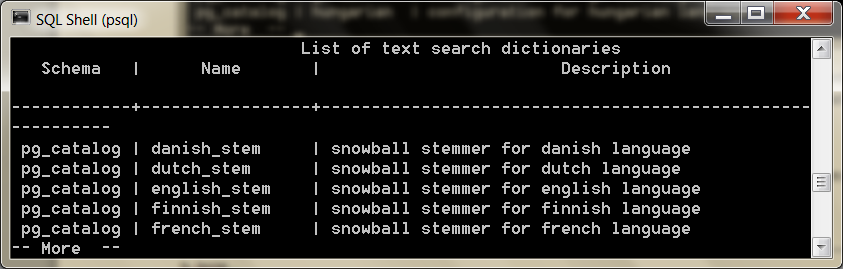
All of the installed configurations can be viewed with this command:

book=# \dF



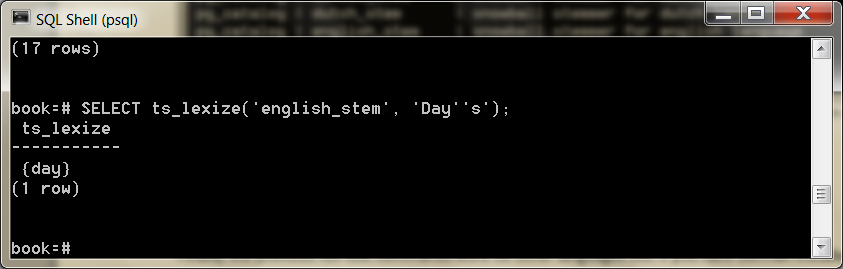
Dictionaries are part of what Postgres uses to generate tsvector lexemes (along with stop words and other tokenizing rules we haven’t covered called *parsers* and *templates*). You can view your system’s list here:

book=# \dFd



You can test any dictionary outright by calling the ts\_lexize() function. Here we find the English stem word of the string *Day’s*.

**SELECT** ts\_lexize(*'english\_stem'*, *'Day''s'*);



Finally, the previous full-text commands work for other languages too. If you have German installed, try this:

**SELECT** to\_tsvector(*'german'*, *'was machst du gerade?'*);



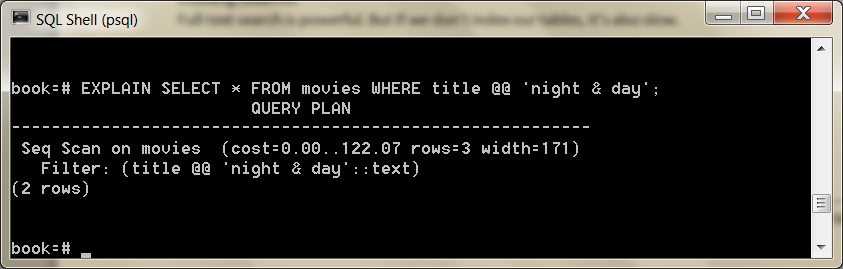
Since *was* (what) and *du* (you) are common, they are marked as stop words in the German dictionary, while *machst* (doing) and *gerade* (now) are stemmed.

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'*;

QUERY PLAN  
---------------------------------------------------------------------------

Seq Scan on movies (cost=10000000000.00..10000000001.12 rows=1 width=68)  
 Filter: (title @@ 'night & day'::text)



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'*;

QUERY PLAN

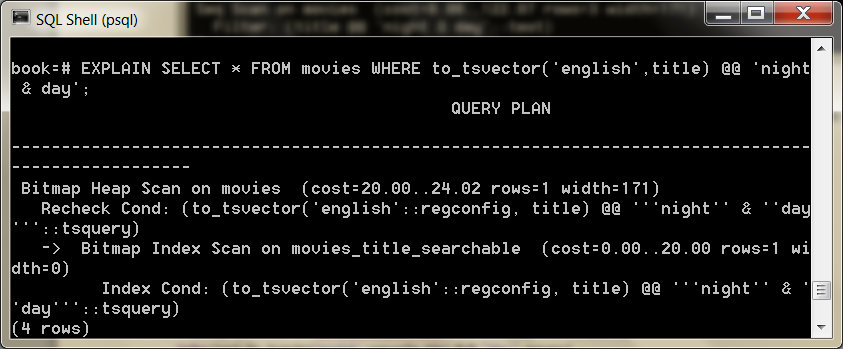
*------------------------------------------------------------------------------------*

Bitmap Heap Scan **on** movies (cost=4.26..8.28 rows=1 width=68)

Recheck Cond: (to\_tsvector(*'english'*::regconfig, title) @@ *'''day'''*::tsquery)

-> Bitmap **Index** Scan **on** movies\_title\_searchable (cost=0.00..4.26 rows=1 width=0)

**Index** Cond: (to\_tsvector(*'english'*::regconfig, title) @@ *'''day'''*::tsquery)



EXPLAIN is important to ensure indexes are used as you expect them. Otherwise,

the index is just wasted overhead.

Metaphones

We’ve inched toward matching less-specific inputs.

LIKE and regular expressions require crafting patterns that can match strings precisely according to their format.

Levenshtein distance allows finding matches that contain minor misspellings but must ultimately be very close to the same string.

Trigrams are a good choice for finding reasonable misspelled matches.

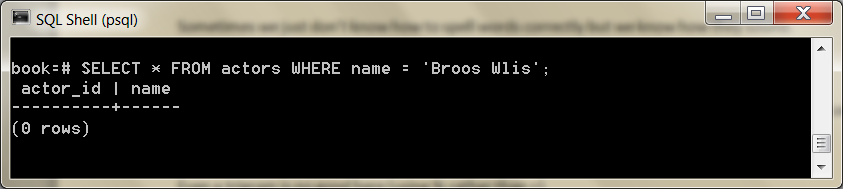
Finally, full-text searching allows natural-language flexibility, in that it can ignore minor words like *a* and *the* and can deal with pluralization.

Sometimes we just don’t know how to spell words correctly but we know how they sound.

We love Bruce Willis and would love to see what movies he’s in.

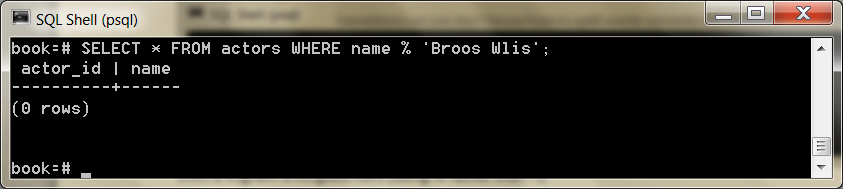
Unfortunately, we can’t remember exactly how to spell his name, so we sound it out as best we can.

**SELECT** \* **FROM** actors **WHERE** name = *'Broos Wlis'*;



Even a trigram is no good here (using % rather than =).

**SELECT** \* **FROM** actors **WHERE** name % *'Broos Wlis'*;



Enter the metaphones, which are algorithms for creating a string representation of word sounds.

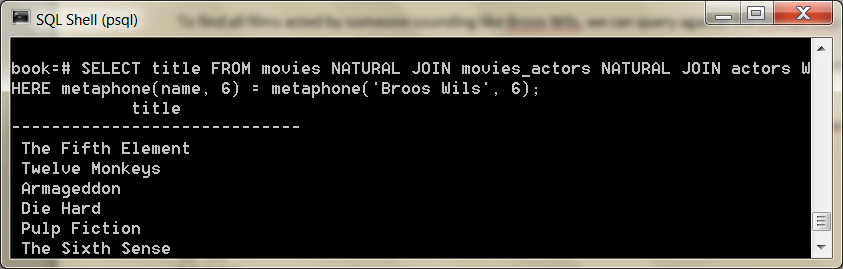
You can define how many characters are in the output string.

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);

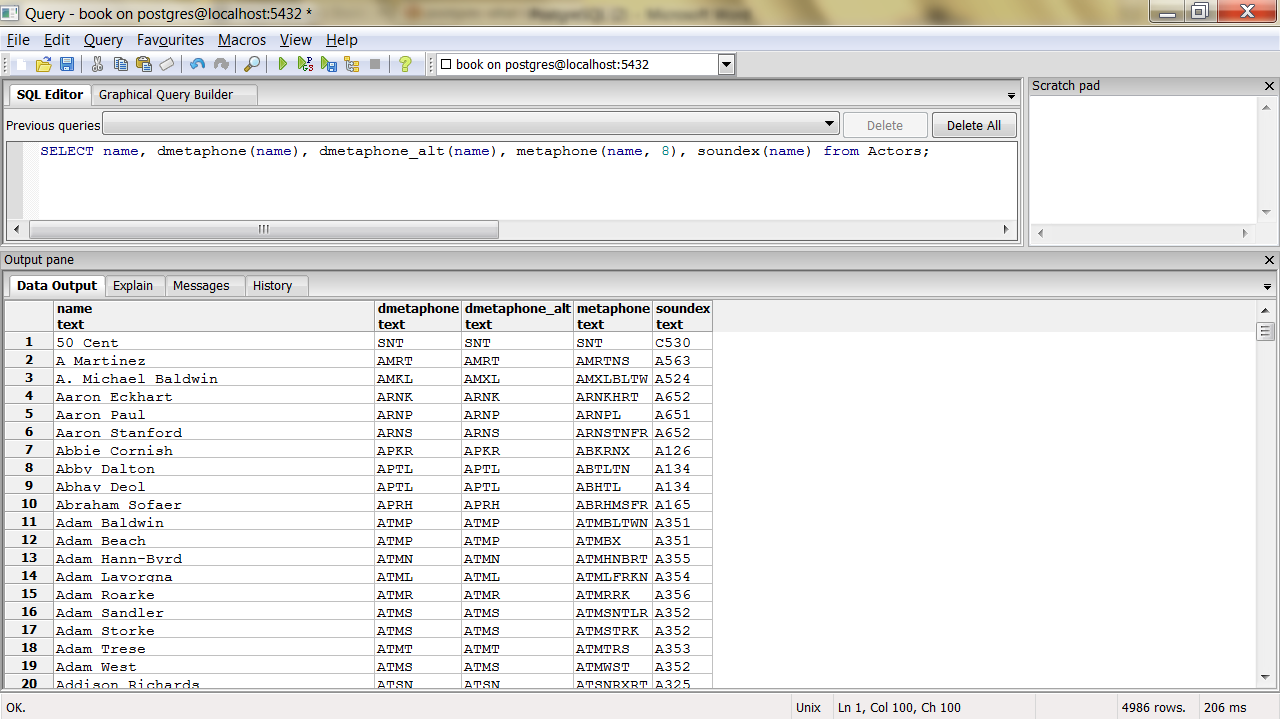


If you peek at the online documentation, you’d see the *fuzzystrmatch* module contains other functions: dmetaphone() (double metaphone), dmetaphone\_alt() (for alternative name pronunciations), and soundex() (a really old algorithm from the 1880s made by the U.S. Census to compare common American surnames).

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

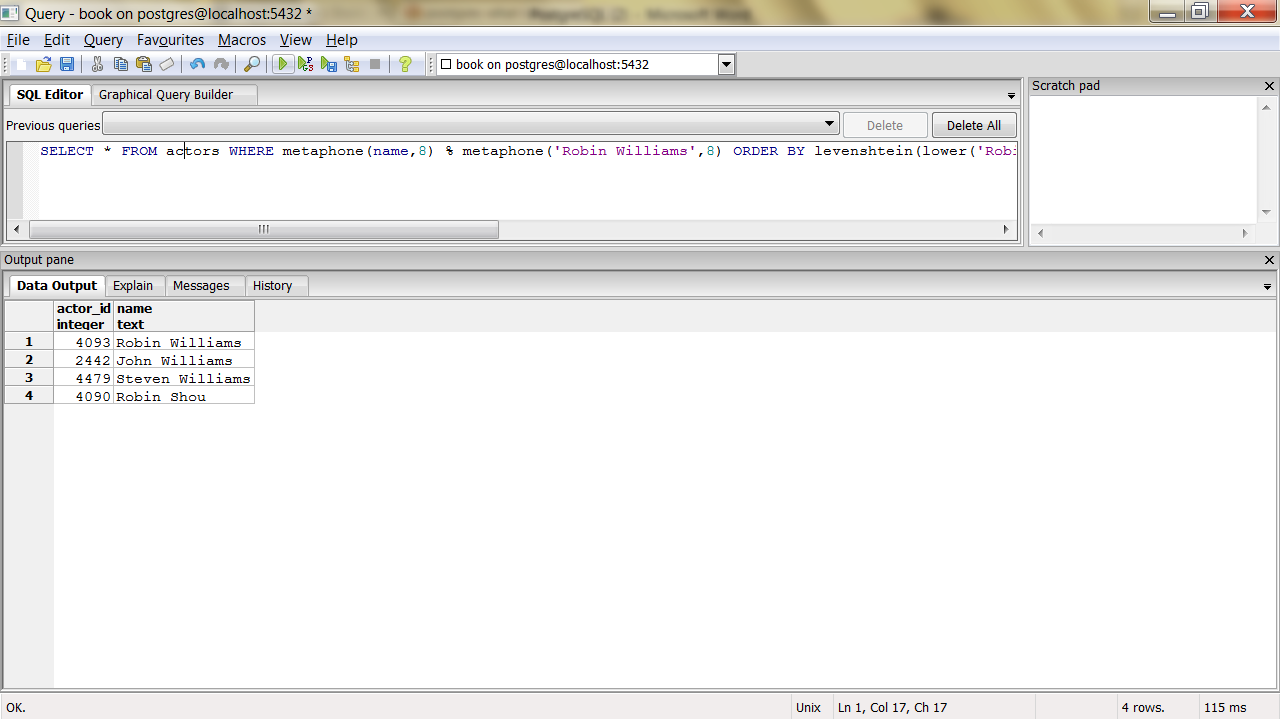
With all of our string searching ducks in a row, we’re ready to start combining them in interesting ways.

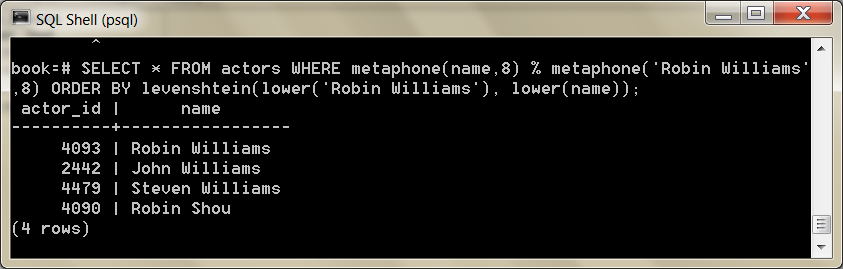
One of the most flexible aspects of metaphones is that their outputs are just strings. This allows you to mix and match with other string matchers.

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.”

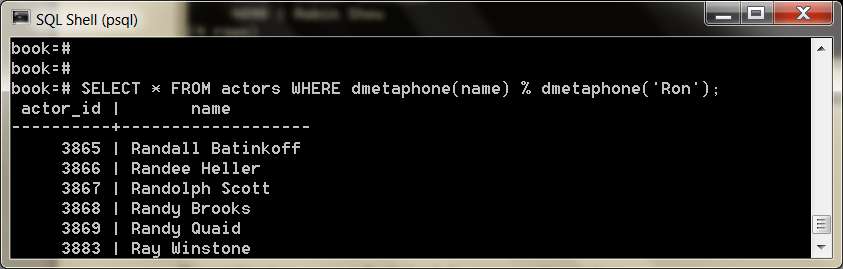
**SELECT** \* **FROM** actors **WHERE** metaphone(name,8) % metaphone(*'Robin Williams'*,8) **ORDER BY** levenshtein(lower(*'Robin Williams'*), lower(name));

or



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'*);

The combinations are vast, limited only by your experimentations.

Genres as a Multidimensional Hypercube

The last contributed package we investigate is cube.

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.

As you may have noticed in the beginning of Day 3, we created a column named genres of type cube.

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

Why represent movie genres as points in n-dimensional space?

Movie categorization is not an exact science, and many movies are not 100 percent comedy or 100 percent tragedy—they are something in between.

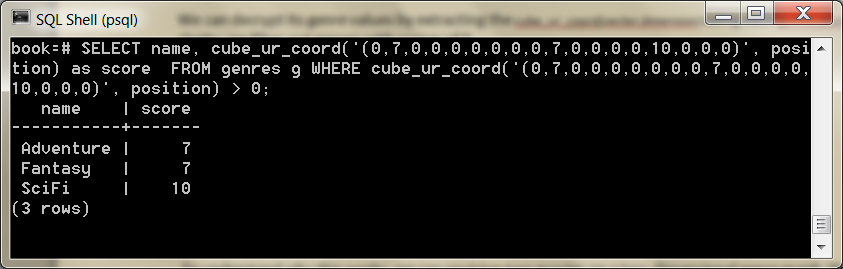
In our system, each genre is scored from (the totally arbitrary numbers) 0 to 10 based on how strong the movie is within that genre—with 0 being nonexistent and 10 being the strongest.

*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).

The genres table describes the position of each dimension in the vector.

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.

To understand why this works, we can envision two movies on a two-dimensional genre graph, like the graph shown below..

If your favorite movie is *Star Wars*, you’ll probably want to see *The Empire Strikes Back* more than *Cries and Whispers*—a story distinctly lacking in adventure.

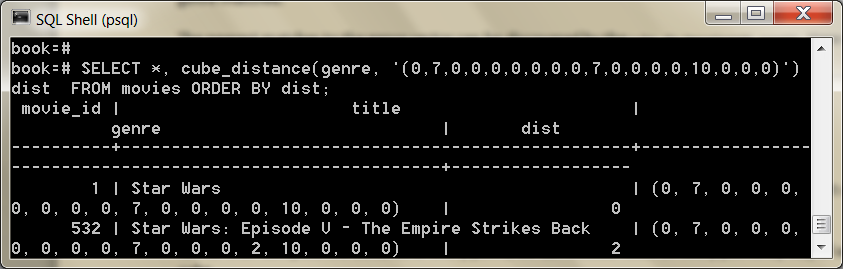
In our two-dimensional universe, it’s a simple nearest-neighbor search to find likely matches.

We can extrapolate this into more dimensions with more genres, be it 2, 3, or 18.

The principle is the same: a nearest-neighbor match to the nearest points in genre space will yield the closest genre matches.

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.

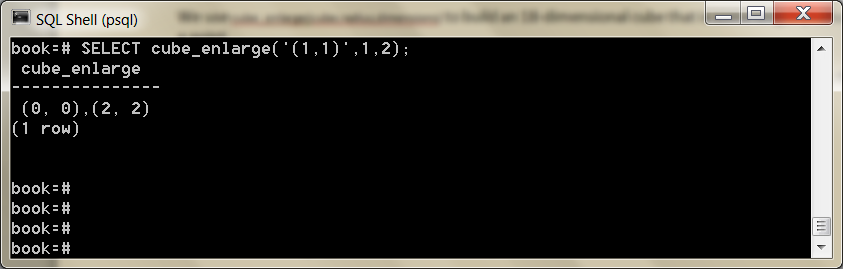
Rather than compute the distance of every point, we can instead focus on likely points by way of a *bounding cube*.

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.

Let’s view a simpler example. If we built a two-dimensional square one unit around a point (1,1), the lower-left point of the square would be at (0,0), and the upper-right point would be (2,2).

**SELECT** cube\_enlarge(*'(1,1)'*,1,2);

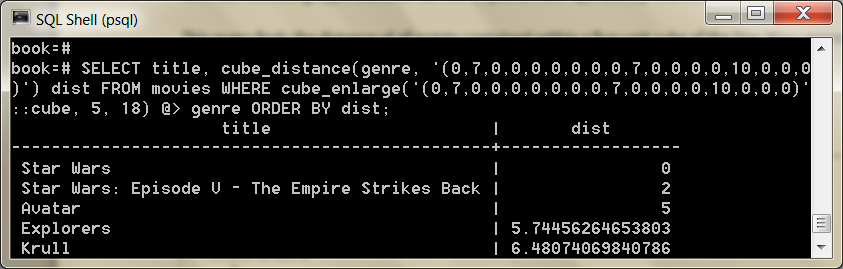


The same principle applies in any number of dimensions.

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

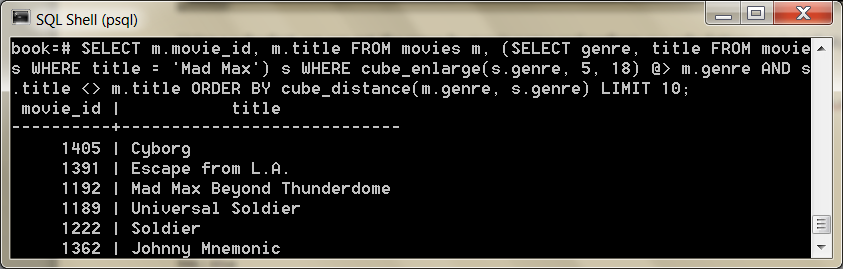
This query finds the distance of all points contained within a five-unit cube of the *Star Wars* genre point.

**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;



This method of movie suggestion is not perfect, but it’s an excellent start.

We will see more dimensional queries in later chapters, such as two-dimensional geographic searches in MongoDB (see *GeoSpatial Queries*).

Wrap-Up

If you haven’t spent much time with relational databases, we highly recommend digging deeper into PostgreSQL, or another relational database, before deciding to scrap it for a newer variety.

Relational databases have been the focus of intense academic research and industrial improvements for more than forty years, and PostgreSQL is one of the top open source relational databases to benefit from these advancements.

PostgreSQL’s Strengths

PostgreSQL’s strengths are as numerous as any relational model: years of research and production use across nearly every field of computing, flexible queryability, and very consistent and durable data.

Most programming languages have battle-tested driver support for Postgres, and many programming models, like object-relational mapping (ORM), assume an underlying relational database.

The crux of the matter is the flexibility of the join.

You needn’t know how you plan to actually query your model, since you can always perform some joins, filters, views, and indexes—odds are good you will always have the ability to extract the data you want.

PostgreSQL is fantastic for what we call “Stepford data” (named for *The Stepford Wives*, a story about a neighborhood where nearly everyone was consistent in style and substance), which is data that is fairly homogeneous and conforms well to a structured schema.

Furthermore, PostgreSQL goes beyond the normal open source RDBMS offerings, such as powerful schema constraint mechanisms.

You can write your own language extensions, customize indexes, create custom datatypes, and even overwrite the parsing of incoming queries.

And where other open source databases may have complex licensing agreements, PostgreSQL is open source in its purest form.

No one owns the code. Anyone can do pretty much anything they want with the project (other than hold authors liable).

The development and distribution are completely community supported.

If you are a fan of free(dom) software you have to respect their general resistance to cashing in on an amazing product.

PostgreSQL’s Weaknesses

Although relational databases are undeniably the most successful style of database over the years, there are cases where it may not be a great fit.

Partitioning is not one of the strong suits of relational databases like PostgreSQL.

If you need to scale out rather than up (multiple parallel datastores rather than a single beefy machine or cluster), you may be better served looking elsewhere.

If your data requirements are too flexible to easily fit into the rigid schema requirements of a relational database or you don’t need the overhead of a full database, require very high-volume reads and writes as key values, or need to store only large blobs of data, then one of the other data- stores might be a better fit.

Parting Thoughts

A relational database is an excellent choice for query flexibility.

While PostgreSQL requires you to design your data up front, it makes no assumptions on how you use that data.

As long as your schema is designed in a fairly normalized way, without duplication or storage of computable values, you should generally be all set for any queries you might need to create.

And if you include the correct modules, tune your engine, and index well, it will perform amazingly well for multiple terabytes of data with very small resource consumption.

Finally, to those for whom data safety is paramount, PostgreSQL’s ACID-compliant transactions ensure your commits are completely atomic, consistent, isolated, and durable.