[Купить книгу The Art of PostgreSQL]

by Dimitri Fonteine

1

Structured Query Language

SQL stands for *Structured Query Language*; the term defines a declarative programming language. As a user, we declare the result we want to obtain in terms of a data processing pipeline that is executed against a known database model and a dataset.

The database model has to be statically declared so that we know the type of every bit of data involved at the time the query is carried out. A query result set defines a relation, of a type determined or inferred when parsing the query.

When working with SQL, as a developer we relatedly work with a type system and a kind of relational algebra. We write code to retrieve and process the data we are interested in, in the specific way we need.

RDBMS and SQL are forcing developers to think in terms of data structure, and to declare both the data structure and the data set we want to obtain via our queries.

Some might then say that SQL forces us to be good developers:

> *I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. Bad programmers worry about the code. Good programmers worry about data structures and their relationships.*
>
> *— Linus Torvalds*

Some of the Code is Written in SQL

If you're reading this book, then it's easy to guess that you are already maintaining at least one application that uses SQL and embeds some SQL queries into its code.

The SQLite project is another implementation of a SQL engine, and one might wonder if it is the Most Widely Deployed Software Module of Any Type?

> *SQLite is deployed in every Android device, every iPhone and iOS device, every Mac, every Windows10 machine, every Firefox, Chrome, and Safari web browser, every installation of Skype, every version of iTunes, every Dropbox client, every TurboTax and QuickBooks, PHP and Python, most television sets and set-top cable boxes, most automotive multimedia systems.*

The page goes on to say that other libraries with similar reach include:

- The original zlib implementation by Jean-loup Gailly and Mark Adler,
- The original reference implementation for *libpng*,
- *Libjpeg* from the Independent JPEG Group.

I can't help but mention that *libjpeg* was developed by Tom Lane, who then contributed to developing the specs of *PNG*. Tom Lane is a Major Contributor to the PostgreSQL project and has been for a long time now. Tom is simply one of the most important contributors to the project.

Anyway, SQL is very popular and it is used in most applications written today. Every developer has seen some `select … from … where …` SQL query string in one form or another and knows some parts of the very basics from SQL-89.

The current SQL standard is SQL:2016 and it includes many advanced data processing techniques. If your application is already using the SQL programming language and SQL engine, then as a developer it's important to fully understand how much can be achieved in SQL, and what service is implemented by this runtime dependency in your software architecture.

Moreover, this service is stateful and hosts all your application user data. In most cases user data as managed by the Relational Database Management Systems that is at the heart of the application code we write, and our code means nothing if we do not have the production data set that delivers value to users.

SQL is a very powerful programming language, and it is a declarative one. It's a wonderful tool to master, and once used properly it allows one to reduce both code size and the development time for new features. This book is written so that you think of good SQL utilization as one of our greatest advantages when writing an application, coding a new business case or implementing a user story!

A First Use Case

Intercontinental Exchange provides a chart with Daily NYSE Group Volume in NYSE Listed, 2017. We can fetch the *Excel* file which is actually a *CSV* file using *tab* as a separator, remove the headings and load it into a PostgreSQL table.

Loading the Data Set

Here's what the data looks like with comma-separated thousands and dollar signs, so we can't readily process the figures as numbers:

```
2010    1/4/2010    1,425,504,460    4,628,115    $38,495,460,645
2010    1/5/2010    1,754,011,750    5,394,016    $43,932,043,406
2010    1/6/2010    1,655,507,953    5,494,460    $43,816,749,660
2010    1/7/2010    1,797,810,789    5,674,297    $44,104,237,184
```

So we create an ad-hoc table definition, and once the data is loaded we then transform it into a proper SQL data type, thanks to *alter table* commands.

```
begin;

drop table if exists factbook;

create table factbook
 (
   year    int,
   date    date,
   shares  text,
   trades  text,
   dollars text
 );

-- datestyle of the database to ISO, MDY
\copy factbook from 'factbook.csv' with delimiter E'\t' null ''

alter table factbook
   alter shares
    type bigint
```

```
    using replace(shares, ',', '')::bigint,

    alter trades
     type bigint
    using replace(trades, ',', '')::bigint,

    alter dollars
     type bigint
    using substring(replace(dollars, ',', '') from 2)::numeric;

commit;
```

We use the PostgreSQL copy functionality to stream the data from the CSV file into our table. The `\copy` variant is a *psql* specific command and initiates *client/server* streaming of the data, reading a local file and sending its content through any established PostgreSQL connection.

Application Code and SQL

Now a classic question is how to list the *factbook* entries for a given month, and because the calendar is a complex beast, we naturally pick February 2017 as our example month.

The following query lists all entries we have in the month of February 2017:

```
\set start '2017-02-01'

  select date,
         to_char(shares, '99G999G999G999') as shares,
         to_char(trades, '99G999G999') as trades,
         to_char(dollars, 'L99G999G999G999') as dollars
    from factbook
   where date >= date :'start'
     and date  < date :'start' + interval '1 month'
order by date;
```

We use the *psql* application to run this query, and *psql* supports the use of variables. The `\set` command sets the *'2017-02-01'* value to the variable *start*, and then we re-use the variable with the expression *:'start'*.

The writing `date :'start'` is equivalent to `date '2017-02-01'` and is called a *decorated literal* expression in PostgreSQL. This allows us to set the data type of the literal value so that the PostgreSQL query parser won't have to guess or infer it from the context.

This first SQL query of the book also uses the *interval* data type to compute the

end of the month. Of course, the example targets February because the end of the month has to be computed. Adding an *interval* value of *1 month* to the first day of the month gives us the first day of the next month, and we use the *less than* (<) strict operator to exclude this day from our result set.

The *to_char()* function is documented in the PostgreSQL section about Data Type Formatting Functions and allows converting a number to its text representation with detailed control over the conversion. The format is composed of *template patterns*. Here we use the following patterns:

- Value with the specified number of digits
- *L*, currency symbol (uses locale)
- *G*, group separator (uses locale)

Other template patterns for numeric formatting are available — see the PostgreSQL documentation for the complete reference.

Here's the result of our query:

```
   date    │    shares     │  trades   │     dollars
═══════════╪═══════════════╪═══════════╪══════════════════
 2017-02-01 │ 1,161,001,502 │ 5,217,859 │ $ 44,660,060,305
 2017-02-02 │ 1,128,144,760 │ 4,586,343 │ $ 43,276,102,903
 2017-02-03 │ 1,084,735,476 │ 4,396,485 │ $ 42,801,562,275
 2017-02-06 │   954,533,086 │ 3,817,270 │ $ 37,300,908,120
 2017-02-07 │ 1,037,660,897 │ 4,220,252 │ $ 39,754,062,721
 2017-02-08 │ 1,100,076,176 │ 4,410,966 │ $ 40,491,648,732
 2017-02-09 │ 1,081,638,761 │ 4,462,009 │ $ 40,169,585,511
 2017-02-10 │ 1,021,379,481 │ 4,028,745 │ $ 38,347,515,768
 2017-02-13 │ 1,020,482,007 │ 3,963,509 │ $ 38,745,317,913
 2017-02-14 │ 1,041,009,698 │ 4,299,974 │ $ 40,737,106,101
 2017-02-15 │ 1,120,119,333 │ 4,424,251 │ $ 43,802,653,477
 2017-02-16 │ 1,091,339,672 │ 4,461,548 │ $ 41,956,691,405
 2017-02-17 │ 1,160,693,221 │ 4,132,233 │ $ 48,862,504,551
 2017-02-21 │ 1,103,777,644 │ 4,323,282 │ $ 44,416,927,777
 2017-02-22 │ 1,064,236,648 │ 4,169,982 │ $ 41,137,731,714
 2017-02-23 │ 1,192,772,644 │ 4,839,887 │ $ 44,254,446,593
 2017-02-24 │ 1,187,320,171 │ 4,656,770 │ $ 45,229,398,830
 2017-02-27 │ 1,132,693,382 │ 4,243,911 │ $ 43,613,734,358
 2017-02-28 │ 1,455,597,403 │ 4,789,769 │ $ 57,874,495,227
(19 rows)
```

The dataset only has data for 19 days in February 2017. Our expectations might be to display an entry for each calendar day and fill it in with either matching data or a zero figure for days without data in our *factbook*.

Here's a typical implementation of that expectation, in Python:

```
#! /usr/bin/env python3

import sys
import psycopg2
import psycopg2.extras
from calendar import Calendar

CONNSTRING = "dbname=yesql application_name=factbook"


def fetch_month_data(year, month):
    "Fetch a month of data from the database"
    date = "%d-%02d-01" % (year, month)
    sql = """
  select date, shares, trades, dollars
    from factbook
   where date >= date %s
     and date  < date %s + interval '1 month'
order by date;
"""
    pgconn = psycopg2.connect(CONNSTRING)
    curs = pgconn.cursor()
    curs.execute(sql, (date, date))

    res = {}
    for (date, shares, trades, dollars) in curs.fetchall():
        res[date] = (shares, trades, dollars)

    return res


def list_book_for_month(year, month):
    """List all days for given month, and for each
    day list fact book entry.
    """
    data = fetch_month_data(year, month)

    cal = Calendar()
    print("%12s | %12s | %12s | %12s" %
          ("day", "shares", "trades", "dollars"))
    print("%12s-+-%12s-+-%12s-+-%12s" %
          ("-" * 12, "-" * 12, "-" * 12, "-" * 12))

    for day in cal.itermonthdates(year, month):
        if day.month != month:
            continue
        if day in data:
            shares, trades, dollars = data[day]
        else:
            shares, trades, dollars = 0, 0, 0
```

```
        print("%12s | %12s | %12s | %12s" %
              (day, shares, trades, dollars))


if __name__ == '__main__':
    year = int(sys.argv[1])
    month = int(sys.argv[2])

    list_book_for_month(year, month)
```

In this implementation, we use the above SQL query to fetch our result set, and moreover to store it in a dictionary. The dict's key is the day of the month, so we can then loop over a calendar's list of days and retrieve matching data when we have it and install a default result set (here, zeroes) when we don't have anything.

Below is the output when running the program. As you can see, we opted for an output similar to the *psql* output, making it easier to compare the effort needed to reach the same result.

```
$ ./factbook-month.py 2017 2
         day |       shares |       trades |      dollars
-------------+--------------+--------------+-------------
  2017-02-01 |   1161001502 |      5217859 |  44660060305
  2017-02-02 |   1128144760 |      4586343 |  43276102903
  2017-02-03 |   1084735476 |      4396485 |  42801562275
  2017-02-04 |            0 |            0 |            0
  2017-02-05 |            0 |            0 |            0
  2017-02-06 |    954533086 |      3817270 |  37300908120
  2017-02-07 |   1037660897 |      4220252 |  39754062721
  2017-02-08 |   1100076176 |      4410966 |  40491648732
  2017-02-09 |   1081638761 |      4462009 |  40169585511
  2017-02-10 |   1021379481 |      4028745 |  38347515768
  2017-02-11 |            0 |            0 |            0
  2017-02-12 |            0 |            0 |            0
  2017-02-13 |   1020482007 |      3963509 |  38745317913
  2017-02-14 |   1041009698 |      4299974 |  40737106101
  2017-02-15 |   1120119333 |      4424251 |  43802653477
  2017-02-16 |   1091339672 |      4461548 |  41956691405
  2017-02-17 |   1160693221 |      4132233 |  48862504551
  2017-02-18 |            0 |            0 |            0
  2017-02-19 |            0 |            0 |            0
  2017-02-20 |            0 |            0 |            0
  2017-02-21 |   1103777644 |      4323282 |  44416927777
  2017-02-22 |   1064236648 |      4169982 |  41137731714
  2017-02-23 |   1192772644 |      4839887 |  44254446593
  2017-02-24 |   1187320171 |      4656770 |  45229398830
  2017-02-25 |            0 |            0 |            0
  2017-02-26 |            0 |            0 |            0
  2017-02-27 |   1132693382 |      4243911 |  43613734358
  2017-02-28 |   1455597403 |      4789769 |  57874495227
```

A Word about SQL Injection

An *SQL Injections* is a security breach, one made famous by the Exploits of a Mom xkcd comic episode in which we read about *little Bobby Tables*.

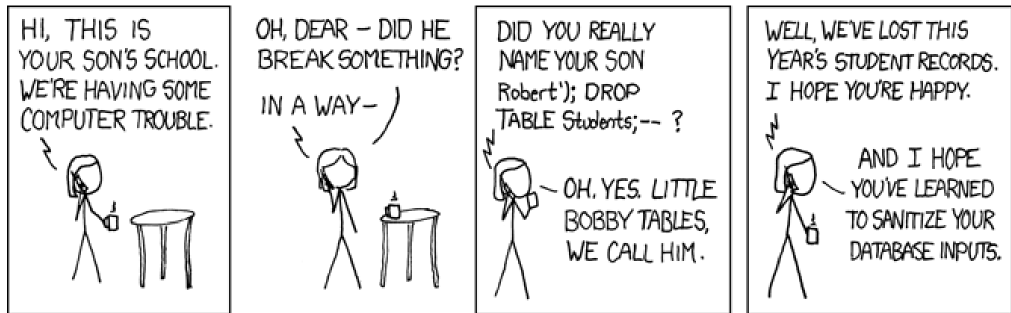

Figure 1.1: Exploits of a Mom

PostgreSQL implements a protocol level facility to send the static SQL query text separately from its dynamic arguments. An SQL injection happens when the database server is mistakenly led to consider a dynamic argument of a query as part of the query text. Sending those parts as separate entities over the protocol means that SQL injection is no longer possible.

The PostgreSQL protocol is fully documented and you can read more about *extended query* support on the Message Flow documentation page. Also relevant is the `PQexecParams` driver API, documented as part of the command execution functions of the `libpq` PostgreSQL C driver.

A lot of PostgreSQL application drivers are based on the libpq C driver, which implements the PostgreSQL protocol and is maintained alongside the main server's code. Some drivers variants also exist that don't link to any C runtime, in which case the PostgreSQL protocol has been implemented in another programming language. That's the case for variants of the JDBC driver, and the `pq` Go driver too, among others.

It is advisable that you read the documentation of your current driver and understand how to send SQL query parameters separately from the main SQL query text; this is a reliable way to never have to worry about *SQL injection* problems ever again.

In particular, *never* build a query string by concatenating your query arguments directly into your query strings, i.e. in the application client code. Never use any

library, ORM or another tooling that would do that. When building SQL query strings that way, you open your application code to serious security risk for no reason.

We were using the psycopg Python driver in our example above, which is based on `libpq`. The documentation of this driver addresses passing parameters to SQL queries right from the beginning.

When using Psycopg the SQL query parameters are interpolated in the SQL query string at the client level. It means you need to trust Psycopg to protect you from any attempt at SQL injection, and we could be more secure than that.

PostgreSQL protocol: server-side prepared statements

It is possible to send the query string and its arguments separately on the wire by using server-side prepared statements. This is a pretty common way to do it, mostly because `PQexecParams` isn't well known, though it made its debut in PostgreSQL 7.4, released November 17, 2003. To this day, a lot of PostgreSQL drivers still don't expose the `PQexecParams` facility, which is unfortunate.

Server-side Prepared Statements can be used in SQL thanks to the PREPARE and EXECUTE commands syntax, as in the following example:

```
prepare foo as
 select date, shares, trades, dollars
   from factbook
  where date >= $1::date
    and date  < $1::date + interval '1 month'
  order by date;
```

And then you can execute the prepared statement with a parameter that way, still at the `psql` console:

```
execute foo('2010-02-01');
```

We then get the same result as before, when using our first version of the Python program.

Now, while it's possible to use the prepare and execute SQL commands directly in your application code, it is also possible to use it directly at the PostgreSQL protocol level. This facility is named Extended Query and is well documented.

Reading the documentation about the protocol implementation, we see the following bits. First the PARSE message:

> In the extended protocol, the frontend first sends a Parse message, which contains a textual query string, optionally some information about data types of parameter placeholders, and the name of a destination prepared-statement object […]

Then, the BIND message:

> Once a prepared statement exists, it can be readied for execution using a Bind message. […] The supplied parameter set must match those needed by the prepared statement.

Finally, to receive the result set the client needs to send a third message, the EXECUTE message. The details of this part aren't relevant now though.

It is very clear from the documentation excerpts above that the query string parsed by PostgreSQL doesn't contain the parameters. The query string is sent in the BIND message. The query parameters are sent in the EXECUTE message. When doing things that way, it is impossible to have SQL injections.

Remember: SQL injection happens when the SQL parser is fooled into believing that a parameter string is in fact a SQL query, and then the SQL engine goes on and executes that SQL statement. When the SQL query string lives in your application code, and the user-supplied parameters are sent separately on the network, there's no way that the SQL parsing engine might get confused.

The following example uses the asyncpg PostgreSQL driver. It's open source and the sources are available at the MagicStack/asyncpg repository, where you can browse the code and see that the driver implements the PostgreSQL protocol itself, and uses server-side prepared statements.

This example is now safe from SQL injection by design, because the server-side prepared statement protocol sends the query string and its arguments in separate protocol messages:

```
import sys
import asyncio
import asyncpg
import datetime
from calendar import Calendar

CONNSTRING = "postgresql://appdev@localhost/appdev?application_name=factbook"

```

```
async def fetch_month_data(year, month):
    "Fetch a month of data from the database"
    date = datetime.date(year, month, 1)
    sql = """
  select date, shares, trades, dollars
    from factbook
   where date >= $1::date
     and date  < $1::date + interval '1 month'
order by date;
"""
    pgconn = await asyncpg.connect(CONNSTRING)
    stmt = await pgconn.prepare(sql)

    res = {}
    for (date, shares, trades, dollars) in await stmt.fetch(date):
        res[date] = (shares, trades, dollars)

    await pgconn.close()

    return res
```

Then, the Python function call needs to be adjusted to take into account the coroutine usage we're now making via asyncio. The function `list_book_for_month` now begins with the following lines:

```
def list_book_for_month(year, month):
    """List all days for given month, and for each
    day list fact book entry.
    """
    data = asyncio.run(fetch_month_data(year, month))
```

The rest of it is as before.

Back to Discovering SQL

Now of course it's possible to implement the same expectations with a single SQL query, without any application code being *spent* on solving the problem:

```
  select cast(calendar.entry as date) as date,
         coalesce(shares, 0) as shares,
         coalesce(trades, 0) as trades,
         to_char(
             coalesce(dollars, 0),
             'L99G999G999G999'
         ) as dollars
    from /*
          * Generate the target month's calendar then LEFT JOIN
```

```
10          * each day against the factbook dataset, so as to have
11          * every day in the result set, whether or not we have a
12          * book entry for the day.
13          */
14         generate_series(date :'start',
15                         date :'start' + interval '1 month'
16                                       - interval '1 day',
17                         interval '1 day'
18         )
19         as calendar(entry)
20         left join factbook
21                on factbook.date = calendar.entry
22 order by date;
```

In this query, we use several basic SQL and PostgreSQL techniques that you might be discovering for the first time:

- SQL accepts comments written either in the `-- comment` style, running from the opening to the end of the line, or C-style with a `/* comment */` style.

  As with any programming language, comments are best used to note our intentions, which otherwise might be tricky to reverse engineer from the code alone.

- *generate_series()* is a PostgreSQL set returning function, for which the documentation reads:

  > Generate a series of values, from start to stop with a step size of step

  As PostgreSQL knows its calendar, it's easy to generate all days from any given month with the first day of the month as a single parameter in the query.

- *generate_series()* is inclusive much like the *BETWEEN* operator, so we exclude the first day of the next month with the expression *- interval '1 day'*.

- The *cast(calendar.entry as date)* expression transforms the generated *calendar.entry*, which is the result of the *generate_series()* function call into the *date* data type.

  We need to *cast* here because the *generate_series()* function returns a set of *timestamp* entries and we don't care about the time parts of it.

- The *left join* in between our generated *calendar* table and the *factbook* table

will keep every calendar row and associate a *factbook* row with it only when the *date* columns of both the tables have the same value.

When the *calendar.date* is not found in *factbook*, the *factbook* columns (*year*, *date*, *shares*, *trades*, and *dollars*) are filled in with *NULL* values instead.

- COALESCE returns the first of its arguments that is not null.

  So the expression *coalesce(shares, 0) as shares* is either how many shares we found in the *factbook* table for this *calendar.date* row, or 0 when we found no entry for the *calendar.date* and the *left join* kept our result set row and filled in the *factbook* columns with *NULL* values.

Finally, here's the result of running this query:

```
    date    │   shares   │ trades  │      dollars
════════════╪════════════╪═════════╪═══════════════════
 2017-02-01 │ 1161001502 │ 5217859 │ $ 44,660,060,305
 2017-02-02 │ 1128144760 │ 4586343 │ $ 43,276,102,903
 2017-02-03 │ 1084735476 │ 4396485 │ $ 42,801,562,275
 2017-02-04 │          0 │       0 │ $               0
 2017-02-05 │          0 │       0 │ $               0
 2017-02-06 │  954533086 │ 3817270 │ $ 37,300,908,120
 2017-02-07 │ 1037660897 │ 4220252 │ $ 39,754,062,721
 2017-02-08 │ 1100076176 │ 4410966 │ $ 40,491,648,732
 2017-02-09 │ 1081638761 │ 4462009 │ $ 40,169,585,511
 2017-02-10 │ 1021379481 │ 4028745 │ $ 38,347,515,768
 2017-02-11 │          0 │       0 │ $               0
 2017-02-12 │          0 │       0 │ $               0
 2017-02-13 │ 1020482007 │ 3963509 │ $ 38,745,317,913
 2017-02-14 │ 1041009698 │ 4299974 │ $ 40,737,106,101
 2017-02-15 │ 1120119333 │ 4424251 │ $ 43,802,653,477
 2017-02-16 │ 1091339672 │ 4461548 │ $ 41,956,691,405
 2017-02-17 │ 1160693221 │ 4132233 │ $ 48,862,504,551
 2017-02-18 │          0 │       0 │ $               0
 2017-02-19 │          0 │       0 │ $               0
 2017-02-20 │          0 │       0 │ $               0
 2017-02-21 │ 1103777644 │ 4323282 │ $ 44,416,927,777
 2017-02-22 │ 1064236648 │ 4169982 │ $ 41,137,731,714
 2017-02-23 │ 1192772644 │ 4839887 │ $ 44,254,446,593
 2017-02-24 │ 1187320171 │ 4656770 │ $ 45,229,398,830
 2017-02-25 │          0 │       0 │ $               0
 2017-02-26 │          0 │       0 │ $               0
 2017-02-27 │ 1132693382 │ 4243911 │ $ 43,613,734,358
 2017-02-28 │ 1455597403 │ 4789769 │ $ 57,874,495,227
(28 rows)
```

When ordering the book package that contains the code and the data set, you can find the SQL queries *02-intro/02-usecase/02.sql* and *02-intro/02-usecase/04.sql*, and the Python script *02-intro/02-usecase/03_factbook-month.py*, and run them against the pre-loaded database *yesql*.

Note that we replaced 60 lines of Python code with a simple enough SQL query. Down the road, that's less code to maintain and a more efficient implementation too. Here, the Python is doing an *Hash Join Nested Loop* where PostgreSQL picks a *Merge Left Join* over two ordered relations. Later in this book, we see how to get and read the PostgreSQL *execution plan* for a query.

Computing Weekly Changes

The analytics department now wants us to add a weekly difference for each day of the result. More specifically, we want to add a column with the evolution as a percentage of the *dollars* column in between the day of the value and the same day of the previous week.

I'm taking the "week over week percentage difference" example because it's both a classic analytics need, though mostly in marketing circles maybe, and because in my experience the first reaction of a developer will rarely be to write a SQL query doing all the math.

Also, computing weeks is another area in which the calendar we have isn't very helpful, but for PostgreSQL taking care of the task is as easy as spelling the word *week*:

```
1   with computed_data as
2   (
3     select cast(date as date)   as date,
4            to_char(date, 'Dy')  as day,
5            coalesce(dollars, 0) as dollars,
6            lag(dollars, 1)
7              over(
8                partition by extract('isodow' from date)
9                    order by date
10             )
11           as last_week_dollars
12      from /*
13            * Generate the month calendar, plus a week before
14            * so that we have values to compare dollars against
15            * even for the first week of the month.
16            */
17           generate_series(date :'start' - interval '1 week',
```