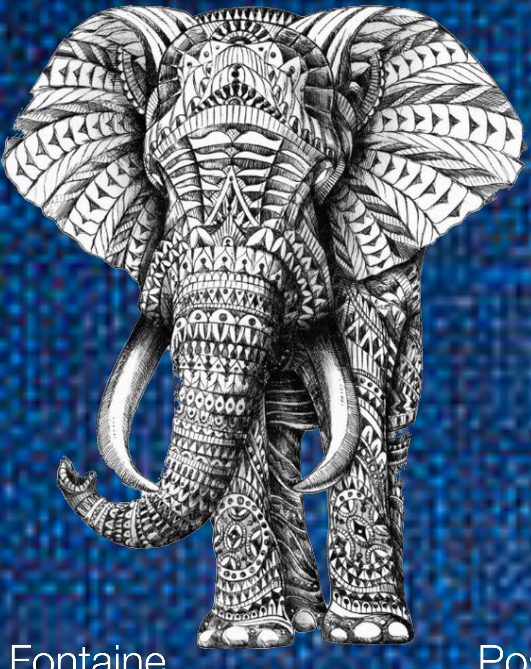
Mastering PostgreSQL

In Application Development



Dimitri Fontaine

PostgreSQL
Major Contributor

1st Edition

Contents

1 Preface

As a developer, *Mastering PostgreSQL in Application Development* is the book you need to read in order to get to the next level of proficiency.

After all, a developer's job encompasses more than just writing code. Our job is to produce results, and for that we have many tools at our disposal. SQL is one of them, and this book teaches you all about it.

PostgreSQL is used to manage data in a centralized fashion, and SQL is used to get exactly the result set needed from the application code. An SQL result set is generally used to fill in-memory data structures so that the application can then process the data. So, let's open this book with a quote about data structures and application code:

> *Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*
>
> — ***Rob Pike***

About the Book

This book is intended for developers working on applications that use a database server. The book specifically addresses the PostgreSQL RDBMS:

it actually is the world's most advanced Open Source database, just like it says in the tagline on the official website. By the end of this book you'll know why, and you'll agree!

I wanted to write this book after having worked with many customers who were making use of only a fraction of what SQL and PostgreSQL are capable of delivering. In most cases, developers I met with didn't know what's possible to achieve in SQL. As soon as they realized — or more exactly, as soon as they were shown what's possible to achieve —, replacing hundreds of lines of application code with a small and efficient SQL query, then in some cases they would nonetheless not know how to integrate a raw SQL query in their code base.

To integrate a SQL query and think about SQL as code, we need to solve what is already solved when using other programming languages: versioning, automated testing, code reviewing, and deployment. Really, this is more about the developer's workflow than the SQL code itself…

In this book, you will learn best practices that help with integrating SQL into your own workflow, and through the many examples provided, you'll see all the reasons why you might be interested in doing more in SQL. Primarily, it means writing fewer lines of code. As Dijkstra said, we should count lines of code as lines spent, so by learning how to use SQL you will be able to spend less to write the same application!

> *The practice is pervaded by the reassuring illusion that programs are just devices like any others, the only difference admitted being that their manufacture might require a new type of craftsmen, viz. programmers. From there it is only a small step to measuring "programmer productivity" in terms of "number of lines of code produced per month". This is a very costly measuring unit because it encourages the writing of insipid code, but today I am less interested in how foolish a unit it is from even a pure business point of view. My point today is that, if we wish to count lines of code, we should not regard them as "lines produced" but as "lines spent": the current conventional wisdom is so foolish as to book that count on the wrong side of the ledger.*
>
> On the cruelty of really teaching computing science, ***Edsger Wybe Dijkstra***, EWD1036

About the Author

Dimitri Fontaine is a PostgreSQL Major Contributor, and has been using and contributing to Open Source Software for the better part of the last twenty years. Dimitri is also the author of the pgloader data loading utility, with fully automated support for database migration from MySQL to PostgreSQL, or from SQLite, or MS SQL…and more.

Dimitri has taken on roles such as developer, maintainer, packager, release manager, software architect, database architect, and database administrator at different points in his career. In the same period of time, Dimitri also started several companies (which are still thriving) with a strong Open Source business model, and he has held management positions as well, including working at the executive level in large companies.

Dimitri runs a blog at http://tapoueh.org with in-depth articles showing advanced use cases for SQL and PostgreSQL.

Acknowledgements

First of all, I'd like to thank all the contributors to the book. I know they all had other priorities in life, yet they found enough time to contribute and help make this book as good as I could ever hope for, maybe even better!

I'd like to give special thanks to my friend ***Julien Danjou*** who's acted as a mentor over the course of writing of the book. His advice about every part of the process has been of great value — maybe the one piece of advice that I most took to the heart has been "write the book you wanted to read".

I'd also like to extend my thanks to the people interviewed for this book. In order of appearance, they are **Yohann Gabory** from the French book "Django Avancé", **Markus Winand** from http://use-the-index-luke.com and http://modern-sql.com, **Grégoire Hubert** author of the PHP POMM project, **Álvaro Hernández Tortosa** who created ToroDB, bringing MongoDB to SQL, and **Kris Jenkins**, functional programmer and author of the YeSQL library for Clojure.

Having insights from SQL users from many different backgrounds has been valuable in achieving one of the major goals of this book: encouraging you, valued readers, to extend your thinking to new horizons. Of course, the horizons I'm referring to include SQL.

I also want to warmly thank the PostgreSQL community. If you've ever joined a PostgreSQL community conference, or even asked questions on the mailing list, you know these people are both incredibly smart and extremely friendly. It's no wonder that PostgreSQL is such a great product as it's produced by an excellent group of well-meaning people who are highly skilled and deeply motivated to solve actual users problems.

Finally, thank you dear reader for having picked this book to read. I hope that you'll have a good time as you read through the many pages, and that you'll learn a lot along the way!

2 Introduction

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 into, 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 run-time dependency in your software architecture.

Moreover, this service is state full 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 coma-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;

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

\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:

```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*.

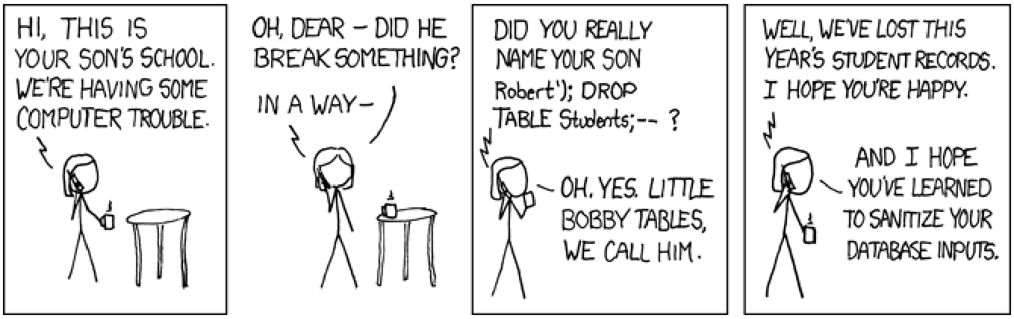

HI, THIS IS YOUR SON'S SCHOOL. WE'RE HAVING SOME COMPUTER TROUBLE.
OH, DEAR – DID HE BREAK SOMETHING?
IN A WAY–
DID YOU REALLY NAME YOUR SON Robert'); DROP TABLE Students;-- ?
OH. YES. LITTLE BOBBY TABLES, WE CALL HIM.
WELL, WE'VE LOST THIS YEAR'S STUDENT RECORDS. I HOPE YOU'RE HAPPY.
AND I HOPE YOU'VE LEARNED TO SANITIZE YOUR DATABASE INPUTS.

Figure 2.1: 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. *Psycopg* is making good use of the functionality we just described, and our `factbook-month.py` program above makes use of the `%s` syntax for SQL query arguments, so we're safe.

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:

```sql
  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
          * each day against the factbook dataset, so as to have
          * every day in the result set, wether or not we have a
          * book entry for the day.
          */
         generate_series(date :'start',
                         date :'start' + interval '1 month'
                                       - interval '1 day',
                         interval '1 day'
         )
         as calendar(entry)
         left join factbook
                on factbook.date = calendar.entry
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*:

```
with computed_data as
(
  select cast(date as date)   as date,
         to_char(date, 'Dy')  as day,
         coalesce(dollars, 0) as dollars,
         lag(dollars, 1)
           over(
             partition by extract('isodow' from date)
                 order by date
           )
         as last_week_dollars
    from /*
          * Generate the month calendar, plus a week before
          * so that we have values to compare dollars against
          * even for the first week of the month.
          */
         generate_series(date :'start' - interval '1 week',
                         date :'start' + interval '1 month'
                                       - interval '1 day',
                         interval '1 day'
         )
         as calendar(date)
         left join factbook using(date)
)
  select date, day,
         to_char(
             coalesce(dollars, 0),
             'L99G999G999G999'
         ) as dollars,
         case when dollars is not null
               and dollars <> 0
              then round(  100.0
                         * (dollars - last_week_dollars)
                         / dollars
                       , 2)
          end
         as "WoW %"
    from computed_data
   where date >= date :'start'
order by date;
```

To implement this case in SQL, we need *window functions* that appeared in the SQL standard in 1992 but are still often skipped in SQL classes. The last thing executed in a SQL statement are *windows functions*, well after *join* operations and *where* clauses. So if we want to see a full week before the first of February, we need to extend our calendar selection a week into the past and then once again restrict the data that we issue to the caller.

That's why we use a *common table expression* — the *WITH* part of the query — to fetch the extended data set we need, including the *last_week_dollars* computed column.

The expression *extract('isodow' from date)* is a standard SQL feature that allows computing the *Day Of Week* following the *ISO* rules. Used as a *partition by* frame clause, it allows a row to be a *peer* to any other row having the same *isodow*. The *lag()* window function can then refer to the previous peer *dollars* value when ordered by date: that's the number with which we want to compare the current *dollars* value.

The *computed_data* result set is then used in the main part of the query as a relation we get data *from* and the computation is easier this time as we simply apply a classic difference percentage formula to the *dollars* and the *last_week_dollars* columns.

Here's the result from running this query:

```
    date    │ day │      dollars      │ WoW %
════════════╪═════╪═══════════════════╪════════
 2017-02-01 │ Wed │ $ 44,660,060,305  │  -2.21
 2017-02-02 │ Thu │ $ 43,276,102,903  │   1.71
 2017-02-03 │ Fri │ $ 42,801,562,275  │  10.86
 2017-02-04 │ Sat │ $               0 │      ¤
 2017-02-05 │ Sun │ $               0 │      ¤
 2017-02-06 │ Mon │ $ 37,300,908,120  │  -9.64
 2017-02-07 │ Tue │ $ 39,754,062,721  │ -37.41
 2017-02-08 │ Wed │ $ 40,491,648,732  │ -10.29
 2017-02-09 │ Thu │ $ 40,169,585,511  │  -7.73
 2017-02-10 │ Fri │ $ 38,347,515,768  │ -11.61
 2017-02-11 │ Sat │ $               0 │      ¤
 2017-02-12 │ Sun │ $               0 │      ¤
 2017-02-13 │ Mon │ $ 38,745,317,913  │   3.73
 2017-02-14 │ Tue │ $ 40,737,106,101  │   2.41
 2017-02-15 │ Wed │ $ 43,802,653,477  │   7.56
 2017-02-16 │ Thu │ $ 41,956,691,405  │   4.26
 2017-02-17 │ Fri │ $ 48,862,504,551  │  21.52
 2017-02-18 │ Sat │ $               0 │      ¤
 2017-02-19 │ Sun │ $               0 │      ¤
 2017-02-20 │ Mon │ $               0 │      ¤
 2017-02-21 │ Tue │ $ 44,416,927,777  │   8.28
 2017-02-22 │ Wed │ $ 41,137,731,714  │  -6.48
 2017-02-23 │ Thu │ $ 44,254,446,593  │   5.19
 2017-02-24 │ Fri │ $ 45,229,398,830  │  -8.03
 2017-02-25 │ Sat │ $               0 │      ¤
 2017-02-26 │ Sun │ $               0 │      ¤
 2017-02-27 │ Mon │ $ 43,613,734,358  │      ¤
 2017-02-28 │ Tue │ $ 57,874,495,227  │  23.25
(28 rows)
```

The rest of the book spends some time to explain the core concepts of *common table expressions* and *window functions* and provides many other examples so that you can master PostgreSQL and issue the SQL queries that fetch exactly the result set your application needs to deal with!

We will also look at the performance and correctness characteristics of issuing more complex queries rather than issuing more queries and doing more of the processing in the application code… or in a Python script, as in the previous example.

Software Architecture

Our first use case in this book allowed us to compare implementing a simple feature in Python and in SQL. After all, once you know enough of SQL, lots of data related processing and presentation can be done directly within your SQL queries. The application code might then be a shell wrapper around a software architecture that is database centered.

In some simple cases, and we'll see more about that in later chapters, it is required for correctness that some processing happens in the SQL query. In many cases, having SQL do the data-related heavy lifting yields a net gain in performance characteristics too, mostly because round-trip times and latency along with memory and bandwidth resources usage depend directly on the size of the result sets.

Mastering PostgreSQL in Application Development focuses on teaching SQL idioms, both the basics and some advanced techniques too. It also contains an approach to database modeling, normalization, and denormalization. That said, it does not address software architecture. The goal of this book is to provide you, the application developer, with new and powerful tools. Determining how and when to use them has to be done in a case by case basis.

Still, a general approach is helpful in deciding how and where to implement application features. The following concepts are important to keep in mind when learning advanced SQL:

- Relational Database Management System

  PostgreSQL is an RDBMS and as such its role in your software architecture is to handle **concurrent access** to **live data** that is manipulated by several applications, or several parts of an application.

  Typically we will find the user-side parts of the application, a front-office and a user back-office with a different set of features depending on the user role, including some kinds of reporting (accounting, finance, analytics), and often some glue scripts here and there, crontabs or the like.

- Atomic, Consistent, Isolated, Durable

At the heart of the concurrent access semantics is the concept of a transaction. A transaction should be **atomic** and **isolated**, the latter allowing for *online backups* of the data.

Additionally, the RDBMS is tasked with maintaining a data set that is **consistent** with the business rules at all times. That's why database modeling and normalization tasks are so important, and why PostgreSQL supports an advanced set of *constraints*.

Durable means that whatever happens PostgreSQL guarantees that it won't lose any *committed* change. Your data is safe. Not even an OS crash is allowed to risk your data. We're left with disk corruption risks, and that's why being able to carry out *online backups* is so important.

- Data Access API and Service

  Given the characteristics listed above, PostgreSQL allows one to implement a data access API. In a world of containers and micro-services, PostgreSQL is the data access service, and its API is SQL.

  If it looks a lot heavier than your typical micro-service, remember that PostgreSQL implements a **stateful service**, on top of which you can build the other parts. Those other parts will be scalable and highly available by design, because solving those problems for *stateless* services is so much easier.

- Structured Query Language

  The data access API offered by PostgreSQL is based on the SQL programming language. It's a **declarative** language where your job as a developer is to describe in detail the *result set* you are interested in.

  PostgreSQL's job is then to find the most efficient way to access only the data needed to compute this result set, and execute the plan it comes up with.

- Extensible (JSON, XML, Arrays, Ranges)

  The SQL language is statically typed: every query defines a new relation that must be fully understood by the system before executing it. That's why sometimes *cast* expressions are needed in your queries.

> PostgreSQL's unique approach to implementing SQL was invented in the 80s with the stated goal of enabling extensibility. SQL operators and functions are defined in a catalog and looked up at run-time. Functions and operators in PostgreSQL support *polymorphism* and almost every part of the system can be extended.
>
> This unique approach has allowed PostgreSQL to be capable of improving SQL; it offers a deep coverage for composite data types and documents processing right within the language, with clean semantics.

So when designing your software architecture, think about PostgreSQL not as *storage* layer, but rather as a *concurrent data access service*. This service is capable of handling data processing. How much of the processing you want to implement in the SQL part of your architecture depends on many factors, including team size, skill set, and operational constraints.

Why PostgreSQL?

While this book focuses on teaching SQL and how to make the best of this programming language in modern application development, it only addresses the PostgreSQL implementation of the SQL standard. That choice is down to several factors, all consequences of PostgreSQL truly being *the world's most advanced open source database*:

- PostgreSQL is open source, available under a BSD like licence named the PostgreSQL licence.
- The PostgreSQL project is done completely in the open, using public mailing lists for all discussions, contributions, and decisions, and the project goes as far as self-hosting all requirements in order to avoid being influenced by a particular company.
- While being developed and maintained in the open by volunteers, most PostgreSQL developers today are contributing in a professional capacity, both in the interest of their employer and to solve real customer problems.
- PostgreSQL releases a new major version about once a year, following a *when it's ready* release cycle.

- The PostgreSQL design, ever since its Berkeley days under the supervision of Michael Stonebraker, allows enhancing SQL in very advanced ways, as we see in the data types and indexing support parts of this book.

- The PostgreSQL documentation is one of the best reference manuals you can find, open source or not, and that's because a patch in the code is only accepted when it also includes editing the parts of the documentations that need editing.

- While new NoSQL systems are offering different trade-offs in terms of operations, guarantees, query languages and APIs, I would argue that PostgreSQL is YeSQL!

In particular, the extensibility of PostgreSQL allows this 20 years old system to keep renewing itself. As a data point, this extensibility design makes PostgreSQL one of the best JSON processing platforms you can find.

It makes it possible to improve SQL with advanced support for new data types even from "userland code", and to integrate processing functions and operators and their indexing support.

We'll see lots of examples of that kind of integration in the book. One of them is a query used in the Schemaless Design in PostgreSQL section where we deal with a Magic The Gathering set of cards imported from a JSON data set:

```sql
select jsonb_pretty(data)
  from magic.cards
 where data @> '{
                "type":"Enchantment",
                "artist":"Jim Murray",
                "colors":["White"]
               }';
```

The `@>` operator reads *contains* and implements JSON searches, with support from a specialized GIN index if one has been created. The *jsonb_pretty()* function does what we can expect from its name, and the query returns *magic.cards* rows that match the JSON criteria for given *type*, *artist* and *colors* key, all as a pretty printed JSON document.

PostgreSQL extensibility design is what allows one to enhance SQL in that way. The query still fully respects SQL rules, there are no tricks here. It is only functions and operators, positioned where we expect them in the *where* clause for the searching and in the *select* clause for the projection that builds the output format.

The PostgreSQL Documentation

This book is not an alternative to the PostgreSQL manual, which in PDF for the 9.6 server weights in at 3376 pages if you choose the A4 format. The table of contents alone in that document includes from pages *iii* to *xxxiv*, that's 32 pages!

This book offers a very different approach than what is expected from a reference manual, and it is in no way to be considered a replacement. Bits and pieces from the PostgreSQL documentation are quoted when necessary, otherwise this book contains lots of links to the reference pages of the functions and SQL commands we utilize in our practical use cases. It's a good idea to refer to the PostgreSQL documentation and read it carefully.

After having spent some time as a developer using PostgreSQL, then as a PostgreSQL contributor and consultant, nowadays I can very easily find my way around the PostgreSQL documentation. Chapters are organized in a logical way, and everything becomes easier when you get used to browsing the reference.

Finally, the *psql* application also includes online help with `\h <sql command>`.

This book does not aim to be a substitute for the PostgreSQL documentation, and other forums and blogs might offer interesting pieces of advice and introduce some concepts with examples. At the end of the day, if you're curious about anything related to PostgreSQL: read the fine manual. No really… this one is fine.

Getting Ready to read this Book

Be sure to use the documentation for the version of PostgreSQL you are using, and if you're not too sure about that just query for it:

```
show server_version;
```

```
 server_version
════════════════
 9.6.5
(1 row)
```

Ideally, you will have a database server to play along with.

- If you're using MacOSX, check out Postgres App to install a PostgreSQL server and the `psql` tool.
- For Windows check https://www.postgresql.org/download/windows/.
- If you're mainly running Linux mainly you know what you're doing already right? My experience is with Debian, so have a look at https://apt.postgresql.org and install the most recent version of PostgreSQL on your station so that you have something to play with locally. For Red Hat packaging based systems, check out https://yum.postgresql.org.

In this book, we will be using `psql` a lot and we will see how to configure it in a friendly way.

You might prefer a more visual tool such as pgAdmin; the key here is to be able to easily edit SQL queries, run them, edit them in order to fix them, see the *explain plan* for the query, etc.

If you have opted for either the *Full Edition* or the *Enterprise Edition* of the book, both include the SQL files. Check out the `toc.txt` file at the top of the files tree, it contains a detailed table of contents and the list of files found in each section, such as in the following example:

```
2 Introduction
  2.1 Some of your code is in SQL
  2.2 A first use case
      02-intro/02-usecase/01_factbook.sql
      02-intro/02-usecase/02.sql
      02-intro/02-usecase/03_factbook-month.py
      02-intro/02-usecase/04.sql
  2.3 Computing Weekly Changes
      02-intro/03-usecase/01.sql
  2.4 PostgreSQL
      02-intro/04-more/01.sql
    2.4.1 The PostgreSQL Documentation
    2.4.2 The setup to read this book
        02-intro/04-more/02_01.sql
```

To run the queries you also need the datasets, and the *Full Edition* includes instructions to fetch the data and load it into your local PostgreSQL instance. The *Enterprise Edition* comes with a PostgreSQL instance containing all the data already loaded for you, and visual tools already setup so that you can click and run the queries.

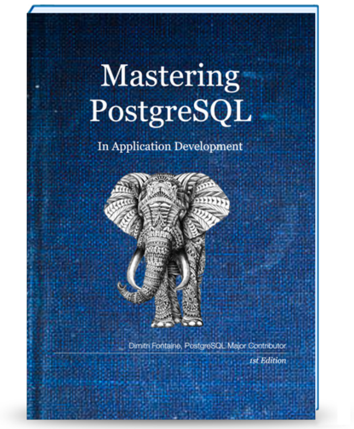
This was Only the Sample...

I hope that you did like the sample! It includes some SQL queries with a typical explaining style as found in the rest of the book.

Mastering PostgreSQL in Application Development includes:

- 8 chapters with 45 sections, as seen in the Table of Contents
- 5 interviews
- 328 pages
- 265 SQL queries
- 12 datasets from the Real World™, with the loading scripts
- 56 tables in 13 schemas
- 4 sample applications, with code written in Python and Go
- Practical examples of data normalization, with queries
- Available in a choice of PDF, ePub and MOBI formats
- Also available in a profesionally printed paperback!

Buy the Book!


Mastering
PostgreSQL
In Application Development
Dimitri Fontaine, PostgreSQL Major Contributor
1st Edition

Now that you've read the sample, and if you liked it, you might be interested into buying the whole book. It's available online at MasteringPostgreSQL.com in several different formats.

You'll find the Book Edition, the Enterprise Edition, the Full Edition and of course the Dead Tree Edition.