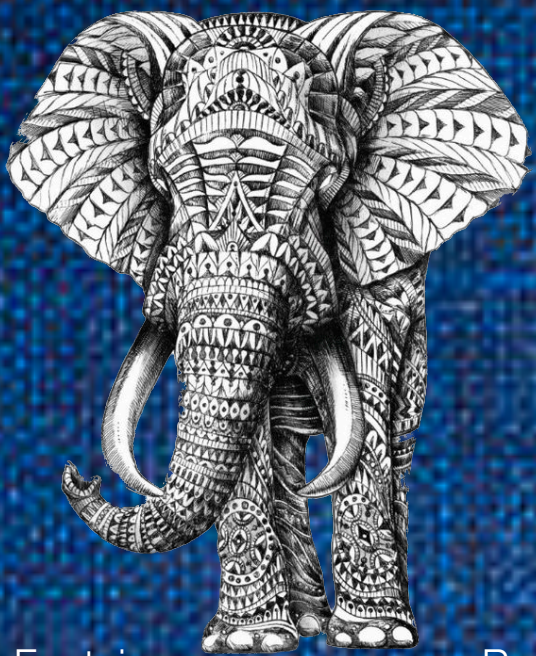
Mastering PostgreSQL

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.

```
1  begin;
2
3  create table factbook
4   (
5     year    int,
6     date    date,
7     shares  text,
8     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:

```
#! /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))
```

```
43
44      for day in cal.itermonthdates(year, month):
45          if day.month != month:
46              continue
47          if day in data:
48              shares, trades, dollars = data[day]
49          else:
50              shares, trades, dollars = 0, 0, 0
51
52          print("%12s | %12s | %12s | %12s" %
53                (day, shares, trades, dollars))
54
55
56  if __name__ == '__main__':
57      year = int(sys.argv[1])
58      month = int(sys.argv[2])
59
60      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*.

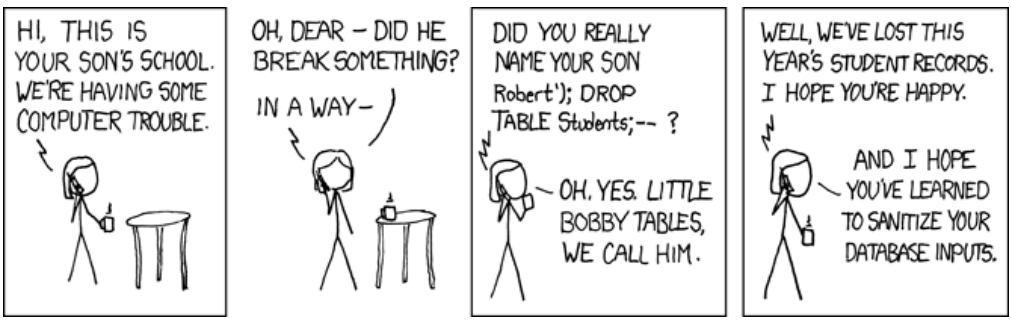


Figure 2.1: Exploits of a Mom

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:

```
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',
```

```
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*:

```
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'
```

```
21          )
22          as calendar(date)
23          left join factbook using(date)
24  )
25    select date, day,
26          to_char(
27              coalesce(dollars, 0),
28              'L99G999G999G999'
29          ) as dollars,
30          case when dollars is not null
31                and dollars <> 0
32               then round(  100.0
33                           * (dollars - last_week_dollars)
34                           / dollars
35                          , 2)
36           end
37          as "WoW %"
38      from computed_data
39     where date >= date :'start'
40  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:

```
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:

```
1 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.

3

Writing Sql Queries

Contents

In this chapter, we are going to learn about how to write SQL queries. There are several ways to accomplish this this, both from the SQL syntax and semantics point of view, and that is going to be covered later. Here, we want to address how to write SQL queries as part of your application code.

Maybe you are currently using an ORM to write your queries and then have never cared about learning how to format, indent and maintain SQL queries. SQL is code, so you need to apply the same rules as when you maintain code written in other languages: indentation, comments, version control, unit testing, etc.

Also to be able to debug what happens in production you need to be able

to easily spot where the query comes from, be able to replay it, edit it, and update your code with the new fixed version of the query.

Before we go into details about the specifics of those concerns, it might be a good idea to review how SQL actually helps you write software, what parts of the code you are writing in the database layer and how much you can or should be writing. The question is this: is SQL a good place to implement business logic?

Next, to get a more concrete example around The Right Way to implement SQL queries in your code, we are going to have a detailed look at a very simple application, so as to work with a specific code base.

After that, we will be able to have a look at those tools and habits that will help you in using SQL in your daily life as an application developer. In particular, this chapter introduces the notion of indexing strategy and explains why this is one of the tasks that the application developer should be doing.

To conclude this part of the book, Yohann Gabory shares his Django expertise with us and covers why SQL is code, which you read earlier in this chapter.

Business Logic

Where to maintain the *business logic* can be a hard question to answer. Each application may be different, and every development team might have a different viewpoint here, from one extreme (all in the application, usually in a *middleware* layer) to the other (all in the database server with the help of stored procedures).

My view is that every SQL query embeds some parts of the business logic you are implementing, thus the question changes from this:

- Should we have business logic in the database?

to this:

- How much of our business logic should be maintained in the database?

The main aspects to consider in terms of where to maintain the business logic are the *correctness* and the *efficiency* aspects of your code architecture and organisation.

Every SQL query embeds some business logic

Before we dive into more specifics, we need to realize that as soon as you send an SQL query to your RDBMS you are already sending *business logic* to the database. My argument is that each and every and all SQL query contains some levels of business logic. Let's consider a few examples.

In the very simplest possible case, you are still expressing some logic in the query. In the Chinook database case, we might want to fetch the list of tracks from a given album:

```
  select name
    from track
   where albumid = 193
order by trackid;
```

What business logic is embedded in that SQL statement?

- The *select* clause only mentions the *name* column, and that's relevant to your application. In the situation in which your application runs this query, the business logic is only interested into the tracks names.
- The *from* clause only mentions the *track* table, somehow we decided that's all we need in this example, and that again is strongly tied to the logic being implemented.
- The *where* clause restricts the data output to the *albumid* 193, which again is a direct translation of our business logic, with the added information that the album we want now is the 193rd one and we're left to wonder how we know about that.
- Finally, the *order by* clause implements the idea that we want to display the track names in the order they appear on the disk. Not only that, it also incorporates the specific knowledge that the *trackid* column ordering is the same as the original disk ordering of the tracks.

A variation on the query would be the following:

```
  select track.name as track, genre.name as genre
    from      track
```

```
3          join genre using(genreid)
4    where albumid = 193
5  order by trackid;
```

This time we add a *join* clause to fetch the genre of each track and choose to return the track name in a column named *track* and the genre name in a column named *genre*. Again, there's only one reason for us to be doing that here: it's because it makes sense with respect to the business logic being implemented in our application.

Granted, those two examples are very simple queries. It is possible to argue that, barring any computation being done to the data set, then we are not actually implementing any *business logic*. It's a fair argument of course. The idea here is that those two very simplistic queries are already responsible for a *part* of the business logic you want to implement. When used as part of displaying, for example, a per album listing page, then it actually is the whole logic.

Let's have a look at another query now. It is still meant to be of the same level of complexity (very low), but with some level of computations being done on-top of the data, before returning it to the main application's code:

```
1    select name,
2           milliseconds * interval '1 ms' as duration,
3           pg_size_pretty(bytes) as bytes
4      from track
5     where albumid = 193
6  order by trackid;
```

This variation looks more like some sort of business logic is being applied to the query, because the columns we sent in the output contain derived values from the server's raw data set.

Business Logic Applies to Use Cases

Up to now, we have been approaching the question from the wrong angle. Looking at a query and trying to decide if it's implementing *business logic* rather than something else (*data access* I would presume) is quite impossible to achieve without a *business case* to solve, also known as a *use case* or maybe even a *user story*, depending on which methodology you are following.

In the following example, we are going to first define a business case we

want to implement, and then we have a look at the SQL statement that we would use to solve it.

Our case is a simple one again: display the list of albums from a given artist, each with its total duration.

Let's write a query for that:

```
  select album.title as album,
         sum(milliseconds) * interval '1 ms' as duration
    from album
         join artist using(artistid)
         left join track using(albumid)
   where artist.name = 'Red Hot Chili Peppers'
group by album
order by album;
```

The output is:

```
          album          |           duration
═══════════════════════╪═══════════════════════════════
 Blood Sugar Sex Magik  | @ 1 hour 13 mins 57.073 secs
 By The Way             | @ 1 hour 8 mins 49.951 secs
 Californication        | @ 56 mins 25.461 secs
(3 rows)
```

What we see here is a direct translation from the business case (or user story if you prefer that term) into a SQL query. The SQL implementation uses joins and computations that are specific to both the data model and the use case we are solving.

Another implementation could be done with several queries and the computation in the application's main code:

1. Fetch the list of albums for the selected artist
2. For each album, fetch the duration of every track in the album
3. In the application, sum up the durations per album

Here's a very quick way to write such an application. It is important to include it here because you might recognize patterns to be found in your own applications, and I want to explain why those patterns should be avoided:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import psycopg2
import psycopg2.extras
import sys
```

```
7  from datetime import timedelta
8
9  DEBUGSQL = False
10 PGCONNSTRING = "user=cdstore dbname=appdev application_name=cdstore"
11
12
13 class Model(object):
14     tablename = None
15     columns = None
16
17     @classmethod
18     def buildsql(cls, pgconn, **kwargs):
19         if cls.tablename and kwargs:
20             cols = ", ".join(['"%s"' % c for c in cls.columns])
21             qtab = '"%s"' % cls.tablename
22             sql = "select %s from %s where " % (cols, qtab)
23             for key in kwargs.keys():
24                 sql += "\"%s\" = '%s'" % (key, kwargs[key])
25             if DEBUGSQL:
26                 print(sql)
27             return sql
28
29
30     @classmethod
31     def fetchone(cls, pgconn, **kwargs):
32         if cls.tablename and kwargs:
33             sql = cls.buildsql(pgconn, **kwargs)
34             curs = pgconn.cursor(cursor_factory=psycopg2.extras.DictCursor)
35             curs.execute(sql)
36             result = curs.fetchone()
37             if result is not None:
38                 return cls(*result)
39
40     @classmethod
41     def fetchall(cls, pgconn, **kwargs):
42         if cls.tablename and kwargs:
43             sql = cls.buildsql(pgconn, **kwargs)
44             curs = pgconn.cursor(cursor_factory=psycopg2.extras.DictCursor)
45             curs.execute(sql)
46             resultset = curs.fetchall()
47             if resultset:
48                 return [cls(*result) for result in resultset]
49
50
51 class Artist(Model):
52     tablename = "artist"
53     columns = ["artistid", "name"]
54
55     def __init__(self, id, name):
56         self.id = id
57         self.name = name
58
```

```
59
60  class Album(Model):
61      tablename = "album"
62      columns = ["albumid", "title"]
63
64      def __init__(self, id, title):
65          self.id = id
66          self.title = title
67          self.duration = None
68
69
70  class Track(Model):
71      tablename = "track"
72      columns = ["trackid", "name", "milliseconds", "bytes", "unitprice"]
73
74      def __init__(self, id, name, milliseconds, bytes, unitprice):
75          self.id = id
76          self.name = name
77          self.duration = milliseconds
78          self.bytes = bytes
79          self.unitprice = unitprice
80
81
82  if __name__ == '__main__':
83      if len(sys.argv) > 1:
84          pgconn = psycopg2.connect(PGCONNSTRING)
85          artist = Artist.fetchone(pgconn, name=sys.argv[1])
86
87          for album in Album.fetchall(pgconn, artistid=artist.id):
88              ms = 0
89              for track in Track.fetchall(pgconn, albumid=album.id):
90                  ms += track.duration
91
92              duration = timedelta(milliseconds=ms)
93              print("%25s: %s" % (album.title, duration))
94      else:
95          print('albums.py <artist name>')
```

Now the result of this code is as following:

```
1  $ ./albums.py "Red Hot Chili Peppers"
2      Blood Sugar Sex Magik: 1:13:57.073000
3                  By The Way: 1:08:49.951000
4            Californication: 0:56:25.461000
```

While you would possibly not write the code in exactly that way, you might be using an application object model which provides a useful set of API entry points and you might be calling object methods that will, in turn, execute the same kind of series of SQL statements. Sometimes, adding insult to injury, your magic object model will insist on hydrating

the intermediate objects with as much information as possible from the database, which translates into `select *` being used. We'll see more about why to avoid `select *` later.

There are several problems related to *correctness* and *efficiency* when this very simple use case is done within several queries, and we're going to dive into them.

Correctness

When using multiple statements, it is necessary to setup the *isolation level* correctly. Also, the connection and transaction semantics of your code should be tightly controlled. Our code snippet here does neither, using a default isolation level setting and not caring much about transactions.

The SQL standard defines four isolation levels and PostgreSQL implements three of them, leaving out *dirty reads*. The isolation level determines which side effects from other transactions your transaction is sensitive to. The PostgreSQL documentation section entitled Transaction Isolation) is quite the reference to read here. If we try and simplify the matter, you can think of the isolation levels like this:

- Read uncommitted

  PostgreSQL accepts this setting and actually implements *read committed* here, which is compliant with the SQL standard;

- Read committed

  This is the default and it allows your transaction to see other transactions changes as soon as they are committed; it means that if you run the following query twice in your transaction but someone else added or removed objects from the stock, you will have different counts at different points in your transaction.

```
1 | SELECT count(*) FROM stock;
```

- Repeatable read

  In this isolation level, your transaction keeps the same *snapshot* of the whole database for its entire duration, from BEGIN to COMMIT.

It is very useful to have that for online backups — a straightforward use case for this feature.

- Serializable

  This level guarantees that a one-transaction-at-a-time ordering of what happens on the server exists with the exact same result as what you're obtaining with concurrent activity.

So by default, we are working in *read committed* isolation level. As most default values, it's a good one when you know how it works and what to expect from it, and more importantly when you should change it.

Each running transaction in a PostgreSQL system can have a different isolation level, so that the online backup tooling may be using *repeatable read* while most of your application is using *read committed*, possibly apart from the stock management facilities which are meant to be *serializable*.

Now, what's happening in our example? Our class `fetch*` methods are all seeing a different database *snapshot*. So what happens to our code if a concurrent user deletes an album from the database in between our *Album.fetchall* call and our *Track.fetchall* call? Or, to make it sound less dramatic, reassigns an album to a different artist to fix some user input error?

What happens is that we'd get a silent empty result set with the impact of showing a duration of 0 to the end-user. In other languages or other spellings of the code, you might have a user-visible error.

Of course, the SQL based solution is immune to those problems: when using PostgreSQL every query always runs within a single consistent snapshot. The isolation level impacts reusing a snapshot from one query to the next.

Efficiency

Efficiency can be measured in a number of ways, including a static and a dynamic analysis of the code written.

The static analysis includes the time it takes a developer to come up with the solution, the maintenance burden it then represents (like the likelihood of bug fixes, the complexity of fixing those bugs), how easy it is to review

the code, etc. The dynamic analysis concerns what happens at runtime in terms of the resources we need to run the code, basically revolving around the processor, memory, network, and disk.

The correct solution here is eight lines of very basic SQL. We may consider that writing this query takes a couple minutes at most and reviewing it is about as easy. To run it from the application side we need to send the query text on the network and we directly retrieve the information we need: for each album its name and its duration. This exchange is done in a single round trip. From the application side, we need to have the list of albums and their duration in memory, and we don't do any computing, so the CPU usage is limited to what needs to be done to talk to the database server and organise the result set in memory, then walk the result it to display it. We must add to that the time it took the server to compute the result for us, and computing the *sum* of the milliseconds is not free.

In the application's code solution, here's what happens under the hood:

- First, we fetch the artist from the database, so that's one network round trip and one SQL query that returns the artist id and its name

  note that we don't need the name of the artist in our use-case, so that's a useless amount of bytes sent on the network, and also in memory in the application.

- Then we do another network round-trip to fetch a list of albums for the artistid we just retrieved in the previous query, and store the result in the application's memory.

- Now for each album (here we only have three of them, the same collection counts 21 albums for *Iron Maiden*) we send another SQL query via the network to the database server and fetch the list of tracks and their properties, including the duration in milliseconds.

- In the same loop where we fetch the tracks durations in milliseconds, we sum them up in the application's memory — we can approximate the CPU usage on the application side to be the same as the one in the PostgreSQL server.

- Finally, the application can output the fetched data.

The thing about picturing the network as a resource is that we now must consider both the latency and the bandwidth characteristics and usage.

That's why in the analysis above the *round trips* are mentioned. In between an application's server and its database, it is common to see latencies in the order of magnitude of 1ms or 2ms.

So from SQL to application's code, we switch from a single network round trips to five of them. That's a lot of extra work for this simple a use case. Here, in my tests, the whole SQL query is executed in less than 1ms on the server, and the whole timing of the query averages around 3ms, including sending the query string and receiving the result set.

With queries running in one millisecond on the server, the network round-trip becomes the main runtime factor to consider. When doing very simple queries against a *primary key* column (`where id = :id`) it's quite common to see execution times around 0.1ms on the server. Which means you could do ten of them in a millisecond... unless you have to wait for ten times for about 1ms for the network transport layer to get the result back to your application's code...

Again this example is a very simple one in terms of *business logic*, still, we can see the cost of avoiding raw SQL both in terms of correctness and efficiency.

Stored Procedures — a Data Access API

When using PostgreSQL it is also possible to create server-side functions. Those SQL objects store code and then execute it when called. The naïve way to create a server-side stored procedure from our current example would be the following:

```
create or replace function get_all_albums
 (
   in  name     text,
   out album    text,
   out duration interval
 )
returns setof record
language sql
as $$
  select album.title as album,
         sum(milliseconds) * interval '1 ms' as duration
    from album
         join artist using(artistid)
         left join track using(albumid)
```

```
   where artist.name = get_all_albums.name
group by album
order by album;
$$;
```

But having to give the name of the artist rather than its *artistid* means that the function won't be efficient to use, and for no good reason. So, instead, we are going to define a better version that works with an artist id:

```
create or replace function get_all_albums
 (
   in  artistid bigint,
   out album    text,
   out duration interval
 )
returns setof record
language sql
as $$
  select album.title as album,
         sum(milliseconds) * interval '1 ms' as duration
    from album
         join artist using(artistid)
         left join track using(albumid)
   where artist.artistid = get_all_albums.artistid
group by album
order by album;
$$;
```

This function is written in *PL/SQL*, so it's basically a SQL query that accepts parameters. To run it, simply do as follows:

```
select * from get_all_albums(127);
```

```
         album          |           duration
════════════════════════╪══════════════════════════
 Blood Sugar Sex Magik  | @ 1 hour 13 mins 57.073 secs
 By The Way             | @ 1 hour 8 mins 49.951 secs
 Californication        | @ 56 mins 25.461 secs
(3 rows)
```

Of course, if you only have the name of the artist you are interested in, you don't need to first do another query. You can directly fetch the *artistid* from a subquery:

```
select *
  from get_all_albums(
         (select artistid
            from artist
           where name = 'Red Hot Chili Peppers')
       );
```

As you can see, the subquery needs its own set of parenthesis even as a function call argument, so we end up with a double set of parenthesis here.

Since PostgreSQL 9.3 and the implementation of the *lateral* join technique, it is also possible to use the function in a join clause:

```
select album, duration
  from artist,
       lateral get_all_albums(artistid)
 where artist.name = 'Red Hot Chili Peppers';
```

```
          album          |           duration
═════════════════════════╪══════════════════════════
 Blood Sugar Sex Magik   | @ 1 hour 13 mins 57.073 secs
 By The Way              | @ 1 hour 8 mins 49.951 secs
 Californication         | @ 56 mins 25.461 secs
(3 rows)
```

Thanks to the *lateral* join, the query is still efficient, and it is possible to reuse it in more complex use cases. Just for the sake of it, say we want to list the album with durations of the artists who have exactly four albums registered in our database:

```
with four_albums as
(
   select artistid
     from album
 group by artistid
   having count(*) = 4
)
  select artist.name, album, duration
    from four_albums
         join artist using(artistid),
         lateral get_all_albums(artistid)
order by artistid, duration desc;
```

Using stored procedure allows reusing SQL code in between use cases, on the server side. Of course, there are benefits and drawbacks to doing so.

Procedural Code and Stored Procedures

The main drawback to using stored procedure is that you must know when to use procedural code or plain SQL with parameters. The previous example can be written in a very ugly way as server-side code:

```
create or replace function get_all_albums
 (
```

```
3     in  name     text,
4     out album    text,
5     out duration interval
6   )
7  returns setof record
8  language plpgsql
9  as $$
10 declare
11   rec record;
12 begin
13   for rec in select albumid
14                from album
15                     join artist using(artistid)
16               where album.name = get_all_albums.name
17   loop
18        select title, sum(milliseconds) * interval '1ms'
19          into album, duration
20          from album
21               left join track using(albumid)
22         where albumid = record.albumid
23      group by title
24      order by title;
25
26     return next;
27   end loop;
28 end;
29 $$;
```

What we see here is basically a re-enactment of everything we said was wrong to do in our application code example. The main difference is that this time, we avoid network round trips, as the loop runs on the database server.

If you want to use stored procedures, please always write them in SQL, and only switch to *PLpgSQL* when necessary. If you want to be efficient, the default should be SQL.

Where to Implement Business Logic?

We saw different ways to implement a very simple use case, with business logic implemented either on the application side, in a SQL query that is part of the application's environment, or as a server-side stored procedure.

The first solution is both incorrect and inefficient, so it should be avoided. It's preferable to exercise PostgreSQL's ability to execute joins rather than play with your network latency. We had five round-trips, with a *ping* of 2

ms, that's 10 ms lost before we do anything else, and we compare that to a query that executes in less than 1 millisecond.

We also need to think in terms of concurrency and scalability. How many concurrent users browsing your album collection do you want to be able to serve? When doing five times as many queries for the same result set, we can imagine that you take a hit of about that ratio in terms of scalability. So rather than invest in an extra layer of caching architecture in front of your APIs, wouldn't it be better to write smarter and more efficient SQL?

As for stored procedures, a lot has already been said. Using them allows the developers to build a data access API in the database server and to maintain it in a transactional way with the database schema: PostgreSQL implements transactions for the *DDL* too. The *DDL* is the *data definition language* which contains the *create*, *alter* and *drop* statements.

Another advantage of using stored procedures is that you send even less data over the network, as the query text is stored on the database server.

A Small Application

Let's write a very basic application where we're going to compare using either classic application code or SQL to solve some common problems. Our goal in this section is to be confronted with managing SQL as part of a code base, and show when to use classic application code or SQL.

Readme First Driven Development

Before writing any code or tests or anything, I like to write the *readme* first. That's this little file explaining to the user why to care for about the application, and maybe some details about how to use it. Let's do that now.

The *cdstore* application is a very simple wrapper on top of the Chinook database. The Chinook data model represents a digital media store, including tables for artists, albums, media tracks, invoices, and customers.

The *cdstore* application allows listing useful information and reports on top of the database, and also provides a way to generate some activity.

Loading the Dataset

When I used the Chinook dataset first, it didn't support PostgreSQL, so I used the SQLite data output, which nicely fits into a small enough data file. Nowadays you will find a PostgreSQL backup file that you can use. It's easier for me to just use pgloader though, so I will just do that.

Another advantage of using pgloader in this book is that we have the following summary output, which lists tables and how many rows we loaded for each of them. This is the first encounter with our dataset.

Here's a truncated output from the pgloader run (edited so that it can fit in the book page format):

```
$ createdb chinook
$ pgloader https://github.com/lerocha/chinook-database/raw/master ⏎
           /ChinookDatabase/DataSources                           ⏎
           /Chinook_Sqlite_AutoIncrementPKs.sqlite
           pgsql://chinook
...
             table name     errors       rows      bytes      total time
---------------------  ---------  ---------  ---------  --------------
                  fetch          0          0                     1.611s
        fetch meta data          0         33                     0.050s
         Create Schemas          0          0                     0.002s
       Create SQL Types          0          0                     0.008s
          Create tables          0         22                     0.092s
         Set Table OIDs          0         11                     0.017s
---------------------  ---------  ---------  ---------  --------------
                 artist          0        275     6.8 kB          0.026s
                  album          0        347    10.5 kB          0.090s
               employee          0          8     1.4 kB          0.034s
                invoice          0        412    31.0 kB          0.059s
              mediatype          0          5     0.1 kB          0.083s
          playlisttrack          0       8715    57.3 kB          0.179s
               customer          0         59     6.7 kB          0.010s
                  genre          0         25     0.3 kB          0.019s
            invoiceline          0       2240    43.6 kB          0.090s
               playlist          0         18     0.3 kB          0.056s
                  track          0       3503   236.6 kB          0.192s
---------------------  ---------  ---------  ---------  --------------
COPY Threads Completion          0          4                     0.335s
         Create Indexes          0         22                     0.326s
 Index Build Completion          0         22                     0.088s
        Reset Sequences          0          0                     0.049s
           Primary Keys          1         11                     0.030s
    Create Foreign Keys          0         11                     0.065s
        Create Triggers          0          0                     0.000s
```

```
       Install Comments          0          0                    0.000s
-----------------------  ---------  ---------  ---------  --------------
      Total import time          ✓      15607   394.5 kB          0.893s
```

Now that the dataset is loaded, we have to fix a badly defined primary key from the SQLite side of things:

```
> \d track
                                Table "public.track"
    Column     |  Type   |                        Modifiers
===============+=========+=========================================================
 trackid       | bigint  | not null default nextval('track_trackid_seq'::regclass)
 name          | text    |
 albumid       | bigint  |
 mediatypeid   | bigint  |
 genreid       | bigint  |
 composer      | text    |
 milliseconds  | bigint  |
 bytes         | bigint  |
 unitprice     | numeric |
Indexes:
    "idx_51519_ipk_track" UNIQUE, btree (trackid)
    "idx_51519_ifk_trackalbumid" btree (albumid)
    "idx_51519_ifk_trackgenreid" btree (genreid)
    "idx_51519_ifk_trackmediatypeid" btree (mediatypeid)

... foreign keys ...

> alter table track add primary key using index idx_51519_ipk_track;
ALTER TABLE
```

Note that as PostgreSQL implements *group by* inference we need this primary key to exists in order to be able to run some of the following queries. This means that as soon as you've loaded the dataset, please fix the primary key so that we are ready to play with the dataset.

Chinook Database

The Chinook database includes basic music elements such as *album*, *artist*, *track*, *genre* and *mediatype* for a music collection. Also, we find the idea of a *playlist* with an association table *playlisttrack*, because any track can take part of several playlists and a single playlist is obviously made of several tracks.

Then there's a model for a customer paying for some tracks with the tables *staff*, *customer*, *invoice* and *invoiceline*.

```
pgloader# \dt chinook.
```

```
            List of relations
 Schema  |      Name       | Type  | Owner
═════════╪═════════════════╪═══════╪═══════
 chinook | album           | table | dim
 chinook | artist          | table | dim
 chinook | customer        | table | dim
 chinook | genre           | table | dim
 chinook | invoice         | table | dim
 chinook | invoiceline     | table | dim
 chinook | mediatype       | table | dim
 chinook | playlist        | table | dim
 chinook | playlisttrack   | table | dim
 chinook | staff           | table | dim
 chinook | track           | table | dim
(11 rows)
```

With that in mind we can begin to explore the dataset with a simple query:

```
  select genre.name, count(*) as count
    from           genre
         left join track using(genreid)
group by genre.name
order by count desc;
```

Which gives us:

```
        name         | count
═════════════════════╪═══════
 Rock                |  1297
 Latin               |   579
 Metal               |   374
 Alternative & Punk  |   332
 Jazz                |   130
 TV Shows            |    93
 Blues               |    81
 Classical           |    74
 Drama               |    64
 R&B/Soul            |    61
 Reggae              |    58
 Pop                 |    48
 Soundtrack          |    43
 Alternative         |    40
 Hip Hop/Rap         |    35
 Electronica/Dance   |    30
 Heavy Metal         |    28
 World               |    28
 Sci Fi & Fantasy    |    26
 Easy Listening      |    24
 Comedy              |    17
 Bossa Nova          |    15
 Science Fiction     |    13
```

```
 Rock And Roll      |    12
 Opera              |     1
(25 rows)
```

Music Catalog

Now, back to our application. We are going to write it in Python, to make it easy to browse the code within the book.

Using the anosql Python library it is very easy to embed SQL code in Python and keep the SQL clean and tidy in `.sql` files. We will look at the Python side of things in a moment.

The `artist.sql` file looks like this:

```
-- name: top-artists-by-album
-- Get the list of the N artists with the most albums
  select artist.name, count(*) as albums
    from           artist
         left join album using(artistid)
group by artist.name
order by albums desc
   limit :n;
```

Having `.sql` files in our source tree allows us to version control them with git, write comments when necessary, and also copy and paste the files between your application's directory and the interactive `psql` shell.

In the case of our `artist.sql` file, we see the use of the *anosql* facility to name variables and we use `limit :n`. Here's how to benefit from that directly in the PostgresQL shell:

```
> \set n 1
> \i artist.sql
    name     | albums
═════════════╪════════
 Iron Maiden |     21
(1 row)

> \set n 3
> \i artist.sql
     name     | albums
══════════════╪════════
 Iron Maiden  |     21
 Led Zeppelin |     14
 Deep Purple  |     11
(3 rows)
```

Of course, you can also set the variable's value from the command line, in case you want to integrate that into *bash* scripts or other calls:

```
psql --variable "n=10" -f artist.sql chinook
```

Albums by Artist

We might also want to include the query from the previous section and that's fairly easy to do now. Our `album.sql` file looks like the following:

```
-- name: list-albums-by-artist
-- List the album titles and duration of a given artist
  select album.title as album,
         sum(milliseconds) * interval '1 ms' as duration
    from album
         join artist using(artistid)
         left join track using(albumid)
   where artist.name = :name
group by album
order by album;
```

Later in this section, we look at the calling Python code.

Top-N Artists by Genre

Let's implement some more queries, such as the Top-N artists per genre, where we sort the artists by their number of appearances in our playlists. This ordering seems fair, and we have a classic Top-N to solve in SQL.

The following extract is our application's `genre-topn.sql` file. The best way to implement a Top-N query in SQL is using a *lateral* join, and the query here is using that technique. We will get back to this kind of join later in the book and learn more details about it. For now, we can simplify the theory down to *lateral join* allowing one to write explicit *loops* in SQL:

```
-- name: genre-top-n
-- Get the N top tracks by genre
select genre.name as genre,
       case when length(ss.name) > 15
            then substring(ss.name from 1 for 15) || '…'
            else ss.name
        end as track,
       artist.name as artist
  from genre
       left join lateral
```

```
11        /*
12         * the lateral left join implements a nested loop over
13         * the genres and allows to fetch our Top-N tracks per
14         * genre, applying the order by desc limit n clause.
15         *
16         * here we choose to weight the tracks by how many
17         * times they appear in a playlist, so we join against
18         * the playlisttrack table and count appearances.
19         */
20        (
21           select track.name, track.albumid, count(playlistid)
22             from           track
23                  left join playlisttrack using (trackid)
24            where track.genreid = genre.genreid
25         group by track.trackid
26         order by count desc
27            limit :n
28        )
29        /*
30         * the join happens in the subquery's where clause, so
31         * we don't need to add another one at the outer join
32         * level, hence the "on true" spelling.
33         */
34            ss(name, albumid, count) on true
35       join album using(albumid)
36       join artist using(artistid)
37 order by genre.name, ss.count desc;
```

Here, we loop through the musical genres we know about, and for each of them, we fetch the `n` tracks with the highest number of appearances in our registered playlists (thanks to the SQL clauses `order by count desc limit :n`). This *correlated subquery* runs for each genre and is *parameterized* with the current `genreid` thanks to the clause `where track.genreid = genre.genreid`. This *where clause* implements the correlation in between the outer loop and the inner one.

Once the inner loop is done in the lateral subquery named `ss` then we join again with the `album` and `artist` tables in order to get the artist name, through the album.

The query may look complex at this stage. The main goal of this book is to help you to find it easier to read and figure out the equivalent code we would have had to write in Python. The main reason why writing moderately complex SQL for this listing is efficiency.

To implement the same thing in application code you have to:

1. Fetch the list of genres (that's one `select name from genre` query)

2. Then for each genre fetch the Top-N list of tracks, which is the `ss` subquery before ran as many times as genres from the application
3. Then for each track selected in this way (that's n times how many genres you have), you can fetch the artist's name.

That's a lot of data to go back and forth in between your application and your database server. It's a lot of useless processing too. So we avoid all this extra work by having the database compute exactly the *result set* we are interested in, and then we have a very simple Python code that only cares about the user interface, here parsing command line options and printing out the result of our queries.

Another common argument against the seemingly complex SQL query is that you know another way to obtain the same result, in SQL, that doesn't involve a *lateral subquery*. Sure, it's possible to solve this Top-N problem in other ways in SQL, but they are all less efficient than the *lateral* method. We will cover how to read an *explain plan* in a later chapter, and that's how to figure out the most efficient way to write a query.

For now, let's suppose this is the best way to write the query. So of course that's the one we are going to include in the application's code, and we need an easy way to then maintain the query.

So here's the whole of our application code:

```
#! /usr/bin/env python3
# -*- coding: utf-8 -*-

import anosql
import psycopg2
import argparse
import sys

PGCONNSTRING = "user=cdstore dbname=appdev application_name=cdstore"

class chinook(object):
    """Our database model and queries"""
    def __init__(self):
        self.pgconn = psycopg2.connect(PGCONNSTRING)
        self.queries = None

        for sql in ['sql/genre-tracks.sql',
                    'sql/genre-topn.sql',
                    'sql/artist.sql',
                    'sql/album-by-artist.sql',
                    'sql/album-tracks.sql']:
```

```
22              queries = anosql.load_queries('postgres', sql)
23              if self.queries:
24                  for qname in queries.available_queries:
25                      self.queries.add_query(qname, getattr(queries, qname))
26              else:
27                  self.queries = queries
28
29      def genre_list(self):
30          return self.queries.tracks_by_genre(self.pgconn)
31
32      def genre_top_n(self, n):
33          return self.queries.genre_top_n(self.pgconn, n=n)
34
35      def artist_by_albums(self, n):
36          return self.queries.top_artists_by_album(self.pgconn, n=n)
37
38      def album_details(self, albumid):
39          return self.queries.list_tracks_by_albumid(self.pgconn, id=albumid)
40
41      def album_by_artist(self, artist):
42          return self.queries.list_albums_by_artist(self.pgconn, name=artist)
43
44
45  class printer(object):
46      "print out query result data"
47
48      def __init__(self, columns, specs, prelude=True):
49          """COLUMNS is a tuple of column titles,
50             Specs an tuple of python format strings
51          """
52          self.columns = columns
53          self.specs = specs
54          self.fstr = " | ".join(str(i) for i in specs)
55
56          if prelude:
57              print(self.title())
58              print(self.sep())
59
60      def title(self):
61          return self.fstr % self.columns
62
63      def sep(self):
64          s = ""
65          for c in self.title():
66              s += "+" if c == "|" else "-"
67          return s
68
69      def fmt(self, data):
70          return self.fstr % data
71
72
73  class cdstore(object):
```

```
74      """Our cdstore command line application. """
75
76      def __init__(self, argv):
77          self.db = chinook()
78
79          parser = argparse.ArgumentParser(
80              description='cdstore utility for a chinook database',
81              usage='cdstore <command> [<args>]')
82          subparsers = parser.add_subparsers(help='sub-command help')
83
84          genres = subparsers.add_parser('genres', help='list genres')
85          genres.add_argument('--topn', type=int)
86          genres.set_defaults(method=self.genres)
87
88          artists = subparsers.add_parser('artists', help='list artists')
89          artists.add_argument('--topn', type=int, default=5)
90          artists.set_defaults(method=self.artists)
91
92          albums = subparsers.add_parser('albums', help='list albums')
93          albums.add_argument('--id', type=int, default=None)
94          albums.add_argument('--artist', default=None)
95          albums.set_defaults(method=self.albums)
96
97          args = parser.parse_args(argv)
98          args.method(args)
99
100     def genres(self, args):
101         "List genres and number of tracks per genre"
102         if args.topn:
103             p = printer(("Genre", "Track", "Artist"),
104                         ("%20s", "%20s", "%20s"))
105
106             for (genre, track, artist) in self.db.genre_top_n(args.topn):
107                 artist = artist if len(artist) < 20 else "%s···" % artist[0:18]
108                 print(p.fmt((genre, track, artist)))
109         else:
110             p = printer(("Genre", "Count"), ("%20s", "%s"))
111             for row in self.db.genre_list():
112                 print(p.fmt(row))
113
114     def artists(self, args):
115         "List genres and number of tracks per genre"
116         p = printer(("Artist", "Albums"), ("%20s", "%5s"))
117         for row in self.db.artist_by_albums(args.topn):
118             print(p.fmt(row))
119
120     def albums(self, args):
121         # we decide to skip parts of the information here
122         if args.id:
123             p = printer(("Title", "Duration", "Pct"),
124                         ("%25s", "%15s", "%6s"))
125             for (title, ms, s, e, pct) in self.db.album_details(args.id):
```

```
126                 title = title if len(title) < 25 else "%s···" % title[0:23]
127                 print(p.fmt((title, ms, pct)))
128
129         elif args.artist:
130             p = printer(("Album", "Duration"), ("%25s", "%s"))
131             for row in self.db.album_by_artist(args.artist):
132                 print(p.fmt(row))
133
134
135 if __name__ == '__main__':
136     cdstore(sys.argv[1:])
```

With this application code and the SQL we saw before we can now run our Top-N query and fetch the single most listed track of each known genre we have in our Chinook database:

```
$ ./cdstore.py genres --topn 1 | head
               Genre |                Track |               Artist
---------------------+----------------------+---------------------
         Alternative |         Hunger Strike |     Temple of the Dog
  Alternative & Punk |         Infeliz Natal |              Raimundos
               Blues |      Knockin On Heav··· |          Eric Clapton
          Bossa Nova |        Onde Anda Você |  Toquinho & Vinícius
           Classical |      Fantasia On Gre··· |  Academy of St. Mar···
              Comedy |       The Negotiation |            The Office
               Drama |            Homecoming |                Heroes
      Easy Listening |      I've Got You Un··· |         Frank Sinatra
```

Of course, we can change our `--topn` parameter and have the top three tracks per genre instead:

```
$ ./cdstore.py genres --topn 3 | head
               Genre |                Track |               Artist
---------------------+----------------------+---------------------
         Alternative |         Hunger Strike |     Temple of the Dog
         Alternative |      Times of Troubl··· |     Temple of the Dog
         Alternative |      Pushin Forward ··· |     Temple of the Dog
  Alternative & Punk |      I Fought The La··· |              The Clash
  Alternative & Punk |         Infeliz Natal |              Raimundos
  Alternative & Punk |             Redundant |             Green Day
               Blues |           I Feel Free |          Eric Clapton
               Blues |      Knockin On Heav··· |          Eric Clapton
```

Now if we want to change our SQL query, for example implementing another way to weight tracks and select the *top* ones per genre, then it's easy to play with the query in `psql` and replace it once you're done.

As we are going to cover in the next section of this book, writing a SQL query happens interactively using a *REPL* tool.

The SQL REPL — An Interactive Setup

PostgreSQL ships with an interactive console with the command line tool named psql. It can be used both for scripting and interactive usage and is moreover quite a powerful tool. Interactive features includes *autocompletion*, *readline* support (history searches, modern keyboard movements, etc), input and output redirection, formatted output, and more.

New users of PostgreSQL often want to find an advanced visual query editing tool and are confused when *psql* is the answer. Most PostgreSQL advanced users and experts don't even think about it and use *psql*. In this chapter, you will learn how to fully appreciate that little command line tool.

Intro to psql

psql implements a REPL: the famous read-eval-print loop. It's one of the best ways to interact with the computer when you're just learning and trying things out. In the case of PostgreSQL you might be discovering a schema, a data set, or just working on a query.

We often see the SQL query when it's fully formed, and rarely get to see the steps that led us there. It's the same with code, most often what you get to see is its final form, not the intermediary steps where the author tries things and refine their understanding of the problem at hand, or the environment in which to solve it.

The process to follow to get to a complete and efficient SQL query is the same as when writing code: iterating from a very simple angle towards a full solution to the problem at hand. Having a *REPL* environment offers an easy way to build up on what you just had before.

The psqlrc Setup

Here we begin with a full setup of *psql* and in the rest of the chapter, we are going to get back to each important point separately. Doing so allows

you to have a fully working environment from the get-go and play around in your PostgreSQL console while reading the book.

```
\set PROMPT1 '%~%x%# '
\x auto
\set ON_ERROR_STOP on
\set ON_ERROR_ROLLBACK interactive

\pset null '¤'
\pset linestyle 'unicode'
\pset unicode_border_linestyle single
\pset unicode_column_linestyle single
\pset unicode_header_linestyle double
set intervalstyle to 'postgres_verbose';

\setenv LESS '-iMFXSx4R'
\setenv EDITOR '/Applications/Emacs.app/Contents/MacOS/bin/emacsclient -nw'
```

Save that setup in the `~/.psqlrc` file, which is read at startup by the *psql* application. As you've already read in the PostgreSQL documentation for *psql*, we have three different settings to play with here:

- `\set [ name [ value [ ... ] ] ]`

  This sets the psql variable name to value, or if more than one value is given, to the concatenation of all of them. If only one argument is given, the variable is set with an empty value. To unset a variable, use the `\unset` command.

- `\setenv name [ value ]`

  This sets the environment variable name to value, or if the value is not supplied, unsets the environment variable.

  Here we use this facility to setup specific environment variables we need from within `psql`, such as the *LESS* setup. It allows invoking the *pager* for each result set but having it take the control of the screen only when necessary.

- `\pset [ option [ value ] ]`

  This command sets options affecting the output of query result tables. *option* indicates which option is to be set. The semantics of *value* vary depending on the selected option. For some options, omitting *value* causes the option to be toggled or unset, as described under the particular option. If no such behavior is mentioned, then omitting *value* just results in the current setting being displayed.

Transactions and psql Behavior

In our case we set several psql variables that change its behavior:

- `\set ON_ERROR_STOP on`

  The name is quite a good description of the option. It allows *psql* to know that it is not to continue trying to execute all your commands when a previous one is throwing an error. It's primarily practical for scripts and can be also set using the command line. As we'll see later, we can easily invoke scripts interactively within our session with the `\i` and `\ir` commands, so the option is still useful to us now.

- `\set ON_ERROR_ROLLBACK interactive`

  This setting changes how *psql* behaves with respect to transactions. It is a very good interactive setup, and must be avoided in batch scripts.

  From the documentation: When set to on, if a statement in a transaction block generates an error, the error is ignored and the transaction continues. When set to interactive, such errors are only ignored in interactive sessions, and not when reading script files. When unset or set to off, a statement in a transaction block that generates an error aborts the entire transaction. The error rollback mode works by issuing an implicit SAVEPOINT for you, just before each command that is in a transaction block, and then rolling back to the savepoint if the command fails.

With the `\set PROMPT1 '%~%x%# '` that we are using, *psql* displays a little star in the prompt when there's a transaction in flight, so you know you need to finish the transaction. More importantly, when you want to type in anything that will have a side effect on your database (modifying the data set or the database schema), then without the star you know you need to first type in `BEGIN`.

Let's see an example output with *ON_ERROR_ROLLBACK* set to off. Here's its default value:

```
f1db# begin;
BEGIN
f1db*# select 1/0;
ERROR:  division by zero
f1db!# select 1+1;
```

```
ERROR:  current transaction is aborted, commands ignored until end of transaction bloc
f1db!# rollback;
ROLLBACK
```

We have an error in our transaction, and we notice that the star prompt is now a flag. The SQL transaction is marked invalid, and the only thing PostgreSQL will now accept from us is to finish the transaction, with either a *commit* or a *rollback* command. Both will result in the same result from the server: `ROLLBACK`.

Now, let's do the same SQL transaction again, this time with *ON_ERROR_ROLLBACK* being set to *interactive*. Now, before each command we send to the server, *psql* sends a savepoint command, which allows it to then issue a rollback to savepoint command in case of an error. This *rollback to savepoint* is also sent automatically:

```
f1db# begin;
BEGIN
f1db*# select 1/0;
ERROR:  division by zero
f1db*# select 1+1;
 ?column?
══════════
        2
(1 row)

f1db*# commit;
COMMIT
```

Notice how this time not only do we get to send successful commands after the error, while still being in a transaction — also we get to be able to *COMMIT* our work to the server.

A Reporting Tool

Getting familiar with *psql* is a very good productivity enhancer, so my advice is to spend some quality time with the documentation of the tool and get used to it. In this chapter, we are going to simplify things and help you to get started.

There are mainly two use cases for *psql*, either as an interactive tool or as a scripting and reporting tool. In the first case, the idea is that you have plenty of commands to help you get your work done, and you can type in SQL right in your terminal and see the result of the query.

In the scripting and reporting use case, you have advanced formatting commands: it is possible to run a query and fetch its result directly in either *asciidoc* or *HTML* for example, given `\pset format`. Say we have a query that reports the N bests known results for a given driver surname. We can use *psql* to set dynamic variables, display tuples only and format the result in a convenient HTML output:

```
~ psql --tuples-only      \
       --set n=1          \
       --set name=Alesi   \
       --no-psqlrc        \
       -P format=html     \
       -d f1db            \
       -f report.sql
```

```
<table border="1">
  <tr valign="top">
    <td align="left">Alesi</td>
    <td align="left">Canadian Grand Prix</td>
    <td align="right">1995</td>
    <td align="right">1</td>
  </tr>
</table>
```

It is also possible to set the connection parameters as environment variables, or to use the same connection strings as in your application's code, so you can test them with copy/paste easily, there's no need to transform them into the `-d dbname -h hostname -p port -U username` syntax:

```
~ psql -d postgresql://dim@localhost:5432/f1db
f1db#

~ psql -d "user=dim host=localhost port=5432 dbname=f1db"
f1db#
```

The query in the `report.sql` file uses the `:'name'` variable syntax. Using `:name` would be missing the quotes around the literal value injected, and `:''` allows one to remedy this even with values containing spaces. *psql* also supports `:"variable"` notation for double-quoting values, which is used for dynamic SQL when identifiers are a parameter (column name or table names).

```
  select surname, races.name, races.year, results.position
    from results
         join drivers using(driverid)
         join races using(raceid)
   where drivers.surname = :'name'
         and position between 1 and 3
```

```
7  order by position
8     limit :n;
```

When running *psql* for reports, it might be good to have a specific setup. In this example, you can see I've been using the `--no-psqlrc` switch to be sure we're not loading my usual interactive setup all with all the UTF-8 bells and whistles, and with *ON_ERROR_ROLLBACK*. Usually, you don't want to have that set for a reporting or a batch script.

You might want to set *ON_ERROR_STOP* though, and maybe some other options.

Discovering a Schema

Let's get back to the interactive features of *psql*. The tool's main task is to send SQL statements to the database server and display the result of the query, and also server notifications and error messages. On top of that *psql* provides a set of client-side commands all beginning with a *backslash* character.

Most of the provided commands are useful for discovering a database schema. All of them are implemented by doing one or several *catalog queries* against the server. Again, it's sending a SQL statement to the server, and it is possible for you to learn how to query the PostgreSQL catalogs by reviewing those queries.

As an example, say you want to report the size of your databases but you don't know where to look for that information. Reading the psql documentation you find that the `\l+` command can do that, and now you want to see the SQL behind it:

```
~# \set ECHO_HIDDEN true
~# \l+
********* QUERY **********
SELECT d.datname as "Name",
       pg_catalog.pg_get_userbyid(d.datdba) as "Owner",
       pg_catalog.pg_encoding_to_char(d.encoding) as "Encoding",
       d.datcollate as "Collate",
       d.datctype as "Ctype",
       pg_catalog.array_to_string(d.datacl, E'\n') AS "Access privileges",
       CASE WHEN pg_catalog.has_database_privilege(d.datname, 'CONNECT')
            THEN pg_catalog.pg_size_pretty(pg_catalog.pg_database_size(d.datname))
            ELSE 'No Access'
       END as "Size",
       t.spcname as "Tablespace",
```

```
       pg_catalog.shobj_description(d.oid, 'pg_database') as "Description"
FROM pg_catalog.pg_database d
  JOIN pg_catalog.pg_tablespace t on d.dattablespace = t.oid
ORDER BY 1;
**************************

List of databases
...
~# \set ECHO_HIDDEN false
```

So now if you only want to have the database name and its on-disk size in bytes, it is as easy as running the following query:

```
  SELECT datname,
         pg_database_size(datname) as bytes
    FROM pg_database
ORDER BY bytes desc;
```

There's not much point in this book including the publicly available documentation of all the commands available in *psql*, so go read the whole manual page to find gems you didn't know about — there are plenty of them!

Interactive Query Editor

You might have noticed that we did set the *EDITOR* environment variable early in this section. This is the command used by *psql* each time you use visual editing commands such as `\e`. This command launches your *EDITOR* on the last edited query (or an empty one) in a temporary file, and will execute the query once you end the editing session.

If you're using *emacs* or *vim* typing with a full-blown editor from within a terminal, it is something you will be very happy to do. In other cases, it is, of course, possible to set *EDITOR* to invoke your favorite IDE if your *psql* client runs locally.

SQL is Code

The first step here is realizing that your database engine actually is part of your application logic. Any SQL statement you write, even the simplest possible, does embed some logic: you are projecting a particular

set of columns, filtering the result to only a part of the available data set (thanks to the where clause), and you want to receive the result in a known ordering. That is already is business logic. Application code is written in SQL.

We compared a simple eight-line SQL query and the typical object model code solving the same use case earlier and analyzed its correctness and efficiency issues. Then in the previous section, we approached a good way to have your SQL queries as .sql files in your code base.

Now that SQL is actually code in your application's source tree, we need to apply the same methodology that you're used to: set a minimum level of expected quality thanks to common indentation rules, code comments, consistent naming, unit testing, and code revision systems.

SQL style guidelines

Code style is mainly about following the *principle of least astonishment* rule. That's why having a clear internal style guide that every developer follows is important in larger teams. We are going to cover several aspects of SQL code style here, from indentation and to alias names.

Indenting is a tool aimed at making it easy to read the code. Let's face it: we spend more time reading code than writing it, so we should always optimize for easy to read the code. SQL is code, so it needs to be properly indented.

Let's see a few examples of bad and good style so that you can decide about your local guidelines.

```
1 SELECT title, name FROM album LEFT JOIN track USING(albumid) WHERE albumid = 1 ORDER
```

Here we have a run-away query all on the same line, making it more difficult than it should for a reader to grasp what the query is all about. Also, the query is using the old habit of all-caps SQL keywords. While it's true that SQL started out a long time ago, we now have color screens and syntax highlighting and we don't write all-caps code anymore... not even in SQL.

My advice is to right align top-level SQL clauses and have them on new lines:

```
  select title, name
    from album left join track using(albumid)
   where albumid = 1
order by 2;
```

Now it's quite a bit easier to understand the structure of this query at a glance and to realize that it is indeed a very basic SQL statement. Moreover, it's easier to spot a problem in the query: *order by 2*. SQL allows one to use output column number as references in some of its clauses, which is very useful at the prompt (because we are all lazy, right?). It makes refactoring harder than it should be though. If we now decide we don't want to output the album's name with each track's row in the result set, as we are actually interested in the track's title and duration, as found in the *milliseconds* column:

```
  select name, milliseconds
    from album left join track using(albumid)
   where albumid = 1
order by 2;
```

So now the ordering has changed, so you need also to change the *order by* clause, obtaining the following diff:

```
@@ -1,4 +1,4 @@
-  select title, name
+  select name, milliseconds
     from album left join track using(albumid)
    where albumid = 1
-order by 2;
+order by 1;
```

This is a very simple example, but nonetheless we can see that the review process now has to take into account why the *order by* clause is modified when what you want to achieve is changing the columns returned.

Now, the right ordering for this query might actually be to return the tracks in the order they appear on the album, which seems to be handled in the Chinook model by the *trackid* itself, so it's better to use that:

```
  select name, milliseconds
    from album left join track using(albumid)
   where albumid = 1
order by trackid;
```

This query is now about to be ready to be checked in into your application's code base, tested and reviewed. An alternative writing would require splitting the from clause into one source relation per line, having the join

appearing more clearly:

```
  select name, milliseconds
    from           album
         left join track using(albumid)
   where albumid = 1
order by trackid;
```

In this style, we see that we indent the join clauses nested in the from clause, because that's the semantics of an SQL query. Also, we left align the table names that take part of the join. An alternative style consists of also entering the join clause (one of either *on* or *using*) in a separate line too:

```
  select name, milliseconds
    from           album
         left join track
             using(albumid)
   where albumid = 1
order by trackid;
```

This extended style is useful when using subqueries, so let's fetch track information from albums we get in a subquery:

```
  select title, name, milliseconds
    from (
           select albumid, title
             from      album
                  join artist using(artistid)
            where artist.name = 'AC/DC'
         )
           as artist_albums
         left join track
             using(albumid)
order by trackid;
```

One of the key things to think about in terms of the style you pick is being consistent. That's why in the previous example we also split the *from* clause in the subquery, even though it's a very simple clause that's not surprising.

SQL requires using parens for subqueries, and we can put that requirement to good use in the way we indent our queries, as shown above.

Another habit that is worth mentioning here consists of writing the join conditions of inner joins in the where clause:

```
SELECT name, title
  FROM artist, album
```

```
 WHERE artist.artistid = album.artistid
   AND artist.artistid = 1;
```

This style reminds us of the 70s and 80s when the SQL standard did specify the join semantics and the join condition. It is extremely confusing to use such a style and doing it is frowned upon. The modern SQL spelling looks like the following:

```
select name, title
  from artist
       inner join album using(artistid)
 where artist.artistid = 1;
```

Here I expanded the inner join to its full notation. The SQL standard introduces *noise words* in the syntax, and both *inner* and *outer* are noise words: a *left*, *right* or *full* join is always an *outer* join, and a straight join always is an *inner* join.

It is also possible to use the *natural join* here, which will automatically expand a join condition over columns having the same name:

```
select name, title
  from artist natural join album
 where artist.artistid = 1;
```

General wisdom dictates that one should avoid *natural joins*: you can (and will) change your query semantics by merely adding a column to or removing a column from a table! In the Chinook model, we have five different tables with a *name* column, none of those being part of the primary key. In most cases, you don't want to join tables on the *name* column...

Because it's fun to do so, let's write a query to find out if the Chinook data set includes cases of a track being named after another artist's, perhaps feflecting their respect or inspiration.

```
  select artist.name as artist,
         inspired.name as inspired,
         album.title as album,
         track.name as track
    from      artist
         join track on track.name = artist.name
         join album on album.albumid = track.albumid
         join artist inspired on inspired.artistid = album.artistid
   where artist.artistid <> inspired.artistid;
```

This gives the following result where we can see two cases of a singer naming a song after their former band's name:

```
    artist      |   inspired    |       album        |     track
================+===============+====================+================
 Iron Maiden    | Paul D'Ianno  | The Beast Live     | Iron Maiden
 Black Sabbath  | Ozzy Osbourne | Speak of the Devil | Black Sabbath
(2 rows)
```

About the query itself, we can see we use the same table twice in the *join* clause, because in one case the artist we want to know about is the one issuing the track in one of their album, and in the other case it's the artist that had their name picked as a track's name. To be able to handle that without confusion, the query uses the SQL standard's relation aliases.

In most cases, you will see very short relation aliases being used. When I typed that query in the *psql* console, I must admit I first picked *a1* and *a2* for artist's relation aliases, because it made it short and easy to type. We can compare such a choice with your variable naming policy. I don't suppose you pass code review when using variable names such as *a1* and *a2* in your code, so don't use them in your SQL query as aliases either.

Comments

The SQL standard comes with two kinds of comments, either per line with the double-dash prefix or per-block delimited with C-style comments using `/* comment */` syntax. Note that contrary to C-style comments, SQL-style comments accept nested comments.

Let's add some comments to our previous query:

```
1  -- artists names used as track names by other artists
2  select artist.name as artist,
3         -- "inspired" is the other artist
4         inspired.name as inspired,
5         album.title as album,
6         track.name as track
7    from      artist
8         /*
9          * Here we join the artist name on the track name,
10         * which is not our usual kind of join and thus
11         * we don't use the using() syntax. For
12         * consistency and clarity of the query, we use
13         * the "on" join condition syntax through the
14         * whole query.
15         */
16        join track
17          on track.name = artist.name
```

```
        join album
          on album.albumid = track.albumid
        join artist inspired
          on inspired.artistid = album.artistid
  where artist.artistid <> inspired.artistid;
```

As with code comments, it's pretty useless to explain what is obvious in the query. The general advice is to give details on what you though was unusual or difficult to write, so as to make the reader's work as easy as possible. The goal of code comments is to avoid ever having to second-guess the *intentions* of the author(s) of it. SQL is code, so we pursue the same goal with SQL.

Comments could also be used to embed the source location where the query comes from in order to make finding it eaiser when we have to debug it in production, should we have to. Given the PostgreSQL's *application_name* facility and a proper use of SQL files in your source code, one can wonder how helpful that technique is.

Unit Tests

SQL is code, so it needs to be tested. The general approach to unit testing code applies beautifully to SQL: given a known input a query should always return the same desired output. That allows you to change your query spelling at will and still check that the alternative still passes your tests.

Examples of query rewriting would include inlining *common table expressions* as sub-queries, expanding *or* branches in a *where* clause as *union all* branches, or maybe using *window function* rather than complex juggling with subqueries to obtain the same result. What I mean here is that there are a lot of ways to rewrite a query while keeping the same semantics and obtaining the same result.

Here's an example of a query rewrite:

```
with artist_albums as
 (
    select albumid, title
      from      album
           join artist using(artistid)
     where artist.name = 'AC/DC'
 )
  select title, name, milliseconds
```

```
    from artist_albums
         left join track
             using(albumid)
order by trackid;
```

The same query may be rewritten with the exact same semantics (but different run-time characteristics) like this:

```
  select title, name, milliseconds
    from (
           select albumid, title
             from      album
                  join artist using(artistid)
            where artist.name = 'AC/DC'
         )
           as artist_albums
         left join track
             using(albumid)
order by trackid;
```

The PostgreSQL project includes many SQL tests to validate its query parser, optimizer and executor. It uses a framework named the *regression tests suite*, based on a very simple idea:

1. Run a SQL file containing your tests with *psql*
2. Capture its output to a text file that includes the queries and their results
3. Compare the output with the expected one that is maintained in the repository with the standard *diff* utility
4. Report any difference as a failure

You can have a look at PostgreSQL repository to see how it's done, as an example we could pick src/test/regress/sql/aggregates.sql and its matching expected result file src/test/regress/expected/aggregates.out.

Implementing that kind of regression testing for your application is quite easy, as the driver is only a thin wrapper around executing standard applications such as *psql* and *diff*. The idea would be to always have a *setup* and a *teardown* step in your SQL test files, wherein the setup step builds a database model and fills it with the test data, and the teardown step removes all that test data.

To automate such a setup and go beyond the obvious, the tool pgTap is a suite of database functions that make it easy to write TAP-emitting unit tests in psql scripts or xUnit-style test functions. The TAP output is suitable for harvesting, analysis, and reporting by a TAP harness, such as

those used in Perl applications.

When using pgTap, see the relation-testing functions for implementing unit tests based on result sets. From the documentation, let's pick a couple example, testing against static result sets as *VALUES*:

```
SELECT results_eq(
    'SELECT * FROM active_users()',
    $$
      VALUES (42, 'Anna'),
             (19, 'Strongrrl'),
             (39, 'Theory')
    $$,
    'active_users() should return active users'
);
```

and *ARRAYS*:

```
SELECT results_eq(
    'SELECT * FROM active_user_ids()',
    ARRAY[ 2, 3, 4, 5]
);
```

As you can see your unit tests are coded in SQL too. This means you have all the SQL power to write tests at your fingertips, and also that you can also check your schema integrity directly in SQL, using PostgreSQL catalog functions.

Straight from the pg_prove command-line tool for running and harnessing pgTAP tests, we can see how it looks:

```
% pg_prove -U postgres tests/
tests/coltap.....ok
tests/hastap.....ok
tests/moretap....ok
tests/pg73.......ok
tests/pktap......ok
All tests successful.
Files=5, Tests=216,  1 wallclock secs
      ( 0.06 usr  0.02 sys +  0.08 cusr  0.07 csys =  0.23 CPU)
Result: PASS
```

You might also find it easy to integrate SQL testing in your current unit testing solution. In Debian and derivatives operating systems, the pg_virtualenv is a tool that creates a temporary PostgreSQL installation that will exist only while you're running your tests.

If you're using Python, read the excellent article from Julien Danjou about databases integration testing strategies with Python where you will learn

more tricks to integrate your database tests using the standard Python toolset.

Your application relies on SQL. You rely on tests to trust your ability to change and evolve your application. You need your tests to cover the SQL parts of your application!

Regression Tests

Regression testing protects against introducing bugs when refactoring code. In SQL too we refactor queries, either because the calling application code is changed and the query must change too, or because we are hitting problems in production and a new optimized version of the query is being checked-in to replace the previous erroneous version.

The way regression testing protects you is by registering the expected results from your queries, and then checking actual results against the expected results. Typically you would run the regression tests each time a query is changed.

The RegreSQL tool implements that idea. It finds SQL files in your code repository and allows registering plan tests against them, and then it compares the results with what's expected.

A typical output from using *RegreSQL* against our *cdstore* application looks like the following:

```
$ regresql test
Connecting to 'postgres:///chinook?sslmode=disable'··· ✓
TAP version 13
ok 1 - src/sql/album-by-artist.1.out
ok 2 - src/sql/album-tracks.1.out
ok 3 - src/sql/artist.1.out
ok 4 - src/sql/genre-topn.top-3.out
ok 5 - src/sql/genre-topn.top-1.out
ok 6 - src/sql/genre-tracks.out
```

In the following example we introduce a bug by changing the test plan without changing the expected result, and here's how it looks then:

```
$ regresql test
Connecting to 'postgres:///chinook?sslmode=disable'··· ✓
TAP version 13
ok 1 - src/sql/album-by-artist.1.out
ok 2 - src/sql/album-tracks.1.out
```

```
# Query File: 'src/sql/artist.sql'
# Bindings File: 'regresql/plans/src/sql/artist.yaml'
# Bindings Name: '1'
# Query Parameters: 'map[n:2]'
# Expected Result File: 'regresql/expected/src/sql/artist.1.out'
# Actual Result File: 'regresql/out/src/sql/artist.1.out'
#
# --- regresql/expected/src/sql/artist.1.out
# +++ regresql/out/src/sql/artist.1.out
# @@ -1,4 +1,5 @@
# -   name     | albums
# -------------+-------
# -Iron Maiden | 21
# +    name     | albums
# +-------------+-------
# +Iron Maiden  | 21
# +Led Zeppelin | 14
#
not ok 3 - src/sql/artist.1.out
ok 4 - src/sql/genre-topn.top-3.out
ok 5 - src/sql/genre-topn.top-1.out
ok 6 - src/sql/genre-tracks.out
```

The diagnostic output allows actions to be taken to fix the problem: either change the expected output (with `regresql update`) or fix the *regresql/plans/src/sql/artist.yaml* file.

A Closer Look

When something wrong happens in production and you want to understand it, one of the important tasks we are confronted with is finding which part of the code is sending a specific query we can see in the monitoring, in the logs or in the interactive activity views.

PostgreSQL implements the *application_name* parameter, which you can set in the connection string and with the *SET* command within your session. It is then possible to have it reported in the server's logs, and it's also part of the system activity view *pg_stat_activity*.

It is a good idea to be quite granular with this setting, going as low as the module or package level, depending on your programming language of choice. It's one of those settings that the main application should have full control of, so usually external (and internal) libs are not setting it.

Indexing Strategy

Coming up with an *Indexing Strategy* is an important step in terms of mastering your PostgreSQL database. It means that you are in a position to make a informed choice about which indexes you need, and most importantly, which you don't need in your application.

A PostgreSQL index allows the system to have new options to find the data your queries need. In the absence of an index, the only option available to your database is a *sequential scan* of your tables. The index *access methods* are meant to be faster than a sequential scan, by fetching the data directly where it is.

Indexing is often though of as a data modeling activity. When using PostgreSQL, some indexes are necessary to ensure data consistency (the C in ACID). Constraints such as *UNIQUE*, *PRIMARY KEY* or *EXCLUDE USING* are only possible to implement in PostgreSQL with a backing index. When an index is used as an implementation detail to ensure data consistency, then the *indexing strategy* is indeed a data modeling activity.

In all other cases, the *indexing strategy* is meant to enable methods for faster access methods to data. Those methods are only going to be exercised in the context of running a SQL query. As writing the SQL queries is the job of a developer, then coming up with the right *indexing strategy* for an application is also the job of the developer.

Indexing for Constraints

When using PostgreSQL some SQL modeling constraints can only be handled with the help of a backing index. That is the case for the primary key and unique constraints, and also for the exclusion constraints created with the PostgreSQL special syntax *EXCLUDE USING*.

In those three constraint cases, the reason why PostgreSQL needs an index is because it allows the system to implement visibility tricks with its MVCC implementation. From the PostgreSQL documentation:

> PostgreSQL provides a rich set of tools for developers to manage concurrent access to data. Internally, data consistency is

> maintained by using a multiversion model (Multiversion Concurrency Control, MVCC). This means that each SQL statement sees a snapshot of data (a database version) as it was some time ago, regardless of the current state of the underlying data. This prevents statements from viewing inconsistent data produced by concurrent transactions performing updates on the same data rows, providing transaction isolation for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock contention in order to allow for reasonable performance in multiuser environments.

If we think about how to implement the *unique* constraint, we soon realize that to be correct the implementation must prevent two concurrent statements from inserting duplicates. Let's see an example with two transactions *t1* and *t2* happening in parallel:

```
t1> insert into test(id) values(1);
t2> insert into test(id) values(1);
```

Before the transactions start the table has no duplicate entry, it is empty. If we consider each transaction, both *t1* and *t2* are correct and they are not creating duplicate entries with the data currently known by PostgreSQL.

Still, we can't accept both the transactions — one of them has to be refused — because they are conflicting with the one another. PostgreSQL knows how to do that, and the implementation relies on the internal code being able to access the indexes in a non-MVCC compliant way: the internal code of PostgreSQL knows what the in-flight non-committed transactions are doing.

The way the internals of PostgreSQL solve this problem is by relying on its index data structure in a non-MVCC compliant way, and this capability is not visible to SQL level users.

So when you declare a *unique* constraint, a *primary key* constraint or an *exclusion constraint* PostgreSQL creates an index for you:

```
> create table test(id integer unique);
CREATE TABLE
Time: 68.775 ms

> \d test
     Table "public.test"
```

```
 Column |  Type   | Modifiers
--------+---------+-----------
 id     | integer |
Indexes:
    "test_id_key" UNIQUE CONSTRAINT, btree (id)
```

And we can see that the index is registered in the system catalogs as being defined in terms of a *constraint*.

Indexing for Queries

PostgreSQL automatically creates only those indexes that are needed for the system to behave correctly. Any and all other indexes are to be defined by the application developers when they need a faster access method to some tuples.

An index cannot alter the result of a query. An index only provides another access method to the data, one that is faster than an sequential scan in most cases. Query semantics and result set don't depend on indexes.

Implementing a user story (or a business case) with the help of SQL queries is the job of the developer. As the author of the SQL statements, the developer also should be responsible for choosing which indexes are needed to support their queries.

Cost of Index Maintenance

An index duplicates data in a specialized format made to optimise a certain type of searches. This duplicated data set is still *ACID* compliant: at *COMMIT;* time, every change that is made it to the main tables of your schema must have made it to the indexes too.

As a consequence, each index adds write costs to your *DML* queries: *insert*, *update* and *delete* now have to maintain the indexes too, and in a transactional way.

That's why we have to define a global *indexing stragegy*. Unless you have infinite IO bandwidth and storage capacity, it is not feasible to index everything in your database.

Choosing Queries to Optmize

In every application, we have some user side parts that require the lowest latency you can provide, and some reporting queries that can run for a little while longer without users complaining.

So when you want to make a query faster and you see that its *explain* plan is lacking index support, think about the query in terms of SLA in your application. Does this query need to run as fast as possible, even when it means that you now have to maintain more indexes?

PostgreSQL Index Access Methods

PostgreSQL implements several index *Access Methods*. An *access method* is a generic algorithm with a clean API that can be implemented for compatible data types. Each algorithm is well adapted to some use cases, which is why it's interesting to maintain several *access methods*.

The PostgreSQL documentation covers index types in the indexes chapter, and tells us that

> PostgreSQL provides several index types: B-tree, Hash, GiST, SP-GiST, GIN and BRIN. Each index type uses a different algorithm that is best suited to different types of queries. By default, the CREATE INDEX command creates B-tree indexes, which fit the most common situations.

Each index access method has been designed to solve specific use case:

- *B-Tree*, or balanced tree

  Balanced indexes are the most common used, by a long shot, because they are very efficient and provide an algorithm that applies to most cases. PostgreSQL implementation of the B-Tree index support is best in class and has been optimized to handle concurrent read and write operations.

  You can read more about the PostgreSQL B-tree algorithm and its theoretical background in the source code file:

  src/backend/access/nbtree/README.

- *GiST*, or generalized search tree

  This access method implements an more general algorithm that again comes from research activities. The GiST Indexing Project from the University of California Berkeley is described in the following terms:

  > The GiST project studies the engineering and mathematics behind content-based indexing for massive amounts of complex content.

  Its implementation in PostgreSQL allows support for 2-dimensional data types such as the geometry *point* or the *ranges* data types. Those data types don't support a total order and as a consequence can't be indexed properly in a B-tree index.

- *SP-GiST*, or spaced partitioned gist

  SP-GiST indexes are the only PostgreSQL index access method implementation that support non-balanced disk-based data structures, such as quadtrees, k-d trees, and radix trees (tries). This is useful when you want to index 2-dimensional data with very different densities.

- *GIN*, or generalized inverted index

  GIN is designed for handling cases where the items to be indexed are composite values, and the queries to be handled by the index need to search for element values that appear within the composite items. For example, the items could be documents, and the queries could be searches for documents containing specific words.

  GIN indexes are "inverted indexes" which are appropriate for data values that contain multiple component values, such as arrays. An inverted index contains a separate entry for each component value. Such an index can efficiently handle queries that test for the presence of specific component values.

  The *GIN* access method is the foundation for the PostgreSQL Full Text Search support.

- *BRIN*, or block range indexes

  BRIN indexes (a shorthand for block range indexes) store summaries about the values stored in consecutive physical block ranges of a ta-

ble. Like GiST, SP-GiST and GIN, BRIN can support many different indexing strategies, and the particular operators with which a BRIN index can be used vary depending on the indexing strategy. For data types that have a linear sort order, the indexed data corresponds to the minimum and maximum values of the values in the column for each block range.

- *Hash*

  Hash indexes can only handle simple equality comparisons. The query planner will consider using a hash index whenever an indexed column is involved in a comparison using the = operator.

  Never use a *hash* index in PostgreSQL before version 10. In PostgreSQL 10 onward, hash index are crash-safe and may be used.

- Bloom filters

  A Bloom filter is a space-efficient data structure that is used to test whether an element is a member of a set. In the case of an index access method, it allows fast exclusion of non-matching tuples via signatures whose size is determined at index creation.

  This type of index is most useful when a table has many attributes and queries test arbitrary combinations of them. A traditional B-tree index is faster than a Bloom index, but it can require many B-tree indexes to support all possible queries where one needs only a single Bloom index. Note however that Bloom indexes only support equality queries, whereas B-tree indexes can also perform inequality and range searches.

  The Bloom filter index is implemented as a PostgreSQL extension starting in PostgreSQL 9.6, and so to be able to use this *access method* it's necessary to first *create extension bloom*.

Both *Bloom* indexes and *BRIN* indexes are mostly useful when covering mutliple columns. In the case of *Bloom* indexes, they are useful when the queries themselves are referencing most or all of those columns in equality comparisons.

Advanced Indexing

The PostgreSQL documentation about indexes covers everything you need to know, in details, including:

- Multicolumn indexes
- Indexes and ORDER BY
- Combining multiple indexes
- Unique indexes
- Indexes on expressions
- Partial indexes
- Partial unique indexes
- Index-only scans

There is of course even more, so consider reading this PostgreSQL chapter in its entirety, as the content isn't repeated in this book, but you will need it to make informed decisions about your indexing strategy.

Adding Indexes

Deciding which indexes to add is central to your *indexing strategy*. Not every query needs to be that fast, and the requirements are mostly user defined. That said, a general system-wide analysis can be achieved thanks to the PostgreSQL extension pg_stat_statements.

Once this PostgreSQL extension is installed and deployed — this needs a PostgreSQL restart — then it's possible to have a list of the most common queries in terms of number of times the query is executed, and the cumulative time it took to execute the query.

You can begin your indexing needs analysis by listing every query that averages out to more than 10 milliseconds, or some other sensible threshold for your application. The only way to understand where time is spent in a query is by using the EXPLAIN command and reviewing the *query plan*. From the documentation of the command:

> PostgreSQL devises a query plan for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to

choose good plans. You can use the EXPLAIN command to see what query plan the planner creates for any query. Plan-reading is an art that requires some experience to master, but this section attempts to cover the basics.

Here's a very rough guide to using *explain* for fixing query performances:

- use the spelling below when using *explain* to understand run time characteristics of your queries:

```
explain (analyze, verbose, buffers)
<query here>;
```

- In particular when you're new to reading *query plans*, use visual tools such as https://explain.depesz.com and PostgreSQL Explain Visualizer, or the once included in pgAdmin.

- First check for row count differences in between the *estimated* and the *effective* numbers.

  Good statistics are critical to the PostgreSQL query planner, and the collected statistics need to be reasonnably up to date. When there's a huge difference in between estimated and effective row counts (several orders of magnitude, a thousand times off or more), check to see if tables are analyzed frequently enough by the Autovacuum Daemon, then check if you should adjust your statistics target,

- Finally, check for time spent doing *sequential scans* of your data, with a *filter* step, as that's the part that a proper index might be able to optimize.

Remember Amdahl's law when optimizing any system: if some step takes 10% of the run time, then the best optimization you can reach from dealing with this step is 10% less, and usually that's by removing the step entirely.

This very rough guide doesn't take into account costly *functions* and *expressions* which may be indexed thanks to *indexes on expressions*, nor *ordering* clauses that might be derived directly from a supporting index.

Query optimisation is a large topic that is not covered in this book, and proper indexing is only a part of it. What this book covers is all the SQL capabilities that you can use to retrieve exactly the result set needed by your application.

The vast majority of slow queries found in the wild are still queries that return way too many rows to the application, straining the network and the servers memory. Returning millions of rows to an application that then displays a summary in a web browser is far too common.

The first rule of optimization in SQL, as is true for code in general, is to answer the following question:

> Do I really need to do any of that?

The very best query optimization technique consists of not having to execute the query at all. Which is why in the next chapter we learn all the SQL functionality that will allow you to execute a single query rather than looping over the result set of a first query only to run an extra query for each row retrieved.

An Interview with Yohann Gabory

Yohann Gabory, Python Django's expert, has published an "Advanced Django" book in France to share his deep understanding of the publication system with Python developers. The book really is a reference on how to use Django to build powerful applications.

As a web backend developer and Django expert, what do you expect from an RDBMS in terms of features and behavior?

> **Consitency and confidence**
>
> *Data is what a web application relies on. You can manage bad quality code but you cannot afford to have data loss or corruption.*
>
> *Someone might say "Hey we do not work for financials, it doesn't matter if we lose some data sometime". What I would answer to this is: if you are ready to lose some data then your data has no value. If your data has no value then there is a big chance that your app has no value either.*
>
> *So let's say you care about your customers and so you care about their data. The first thing you must guaranty is confi-*

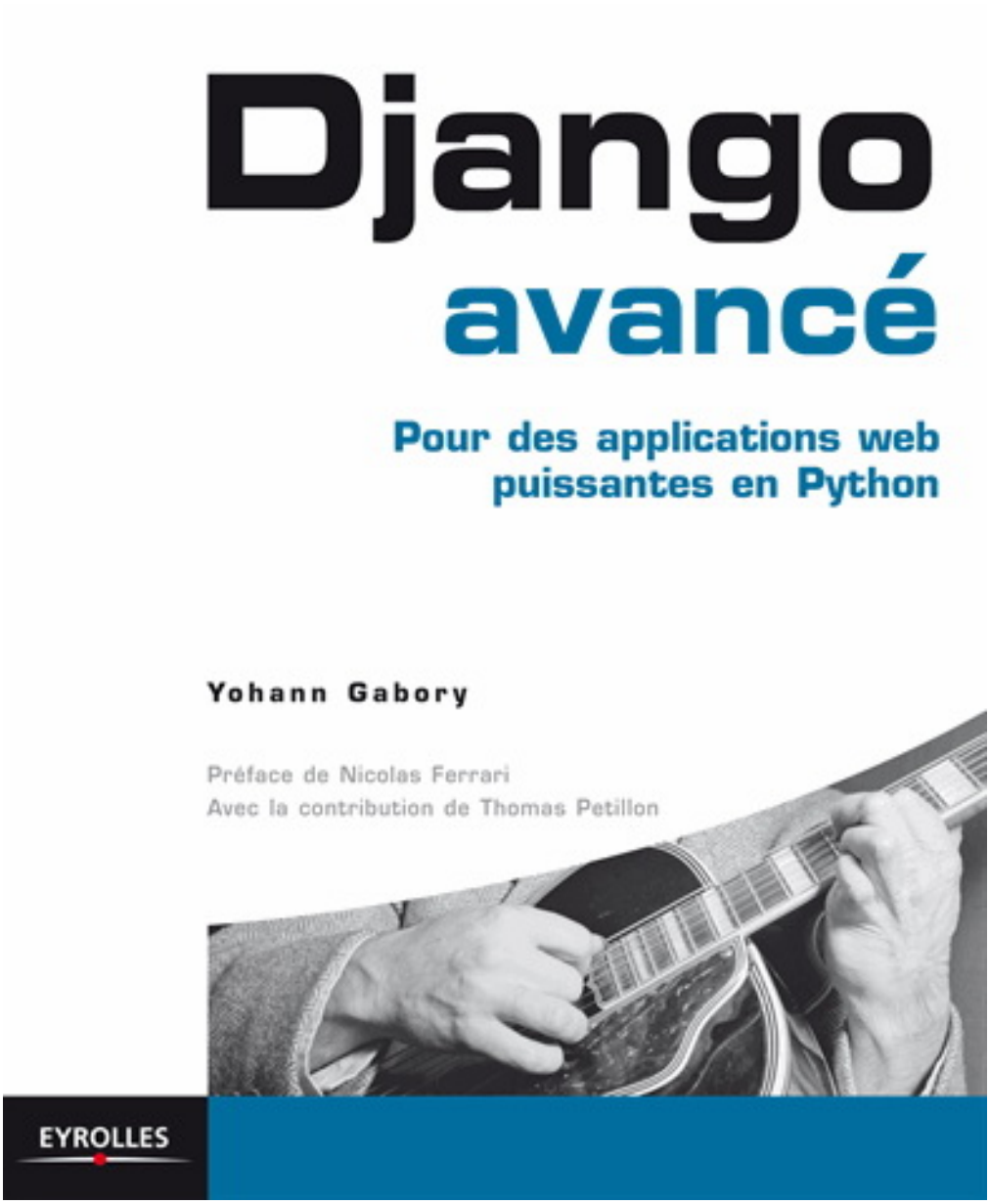


Figure 3.1: Advanced Django

dence. Your users must trust you when you say, "I have saved your data". They must trust you when you say, "Your data is not corrupted".

So what is the feature I first expect?

Don't mess up my database with invalid or corrupted data. Ensure that when my database says something is saved, it really is.

Code in SQL

Of course, this means that each time the coherence of my database is involved I do not rely on my framework or my Python code. I rely on SQL code.

I need my database to be able to handle code within itself — procedure, triggers, check_constraints — those are the most basic features I need from a database.

Flexible when I want, rigid when I ask

As a developer when first implementing a proof of concept or a MVC you cannot ask me to know perfectly how I will handle my data in the future. Some information that does not seem very revelant will be mandatory or something else I tough was mandatory is not after all.

So I need my database to be flexible enough to let me easily change what is mandatory and what is not.

This point is the main reason some developers fly to NoSQL databases. Because they see the schemaless *options as a way to not carefully specify their database schema.*

At first sight this can seem like a good idea. In fact, this is a terrible one. Because tomorrow you will need consistency and non-permissive schema. When it happens, you will be on your own, lost in a world of inconsistency, corrupted data and "eventually consistent" records.

I will not talk about writing consistency and relational checks in code because it reminds me of nightmares called race-conditions *and* Heisenbugs.

What I really expect from my RDBMS is to let me begin schemaless and after some time, let me specify mandatory fields, relation insurance and so on. If you think I'm asking too much, have a look at jsonb *or* hstore.

What makes you want to use PostgreSQL rather than something else in your Django projects? Are there any difficulties to be aware of when using PostgreSQL?

Django lets you use a lot of different databases. You can use SQLite, MariaDB, PostgreSQL and some others. Of course, you can expect from some databases availability, consistency, isolation, and durability. This allows you to make decent applications. But there is always a time where you need more. Expecially some database type that could match Python type. Think about list, dictionary, ranges, timestamp, timezone, date and datetime.

All of this (and more) can be found in PostgreSQL. This is so true that there are now in Django some specific models fields (the Django representation of a column) to handle those great PostgreSQL fields.

When it comes to choosing a database why someone wants to use something other than the most full-featured?

But don't think I choose PostgreSQL only for performance, easiness of use and powerful features. It's also a really warm place to code with confidence.

Because Django has a migration management system that can handle pure SQL I can write advanced SQL functions and triggers directly in my code. Those functions can use the most advanced features of PostgreSQL and stay right in front of me, in my Git, easily editable.

In fact version after version, Django let you use your database more and more. You can now use SQL function like COALESCE, NOW, aggregation functions and more directly in your Django code. And those function you write are plain SQL.

This also means that version after version your RDBMS choice is more and more important. Do you want to choose a tool that

can do half the work you expect from it?

Me neither.

Django comes with an internal ORM that maps an object model to a relational table and allows it to handle "saving" objects and SQL query writing. Django also supports raw SQL. What is your general advice around using the ORM?

Well this is a tough question. Some will says ORM sucks. Some others says mixing SQL and Python code in your application is ugly. I think they are both right. Of course, an ORM limits you a lot. Of course writing SQL everytime you need to talk to your database is not sustainable in the long run.

When your queries are so simple you can express them with your ORM why not use it? It will generate a SQL query as good as anybody could write. It will hydrate a Django object you can use right away, in a breeze.

Think about:

```
MyModel.objects.get(id=1)
```

This is equivalent to:

```
select mymodel.id, mymodel.other_field, ...
  from mymodel
 where id=1;
```

Do you think you could write better SQL?

ORM can manage all of your SQL needs. There is also some advice to avoid the N+1 dilemma. The aggregation system relies on SQL and is fairly decent.

But if you don't pay attention, it will bite you hard.

The rule of thumb for me is to never forget what your ORM is meant for: translate SQL records into Python objects.

If you think it can handle anything more, like avoiding writing SQL, managing indexes etc... you are wrong.

The main Django ORM philosophy is to let you drive the car.

- *First always be able to translate your ORM query into the SQL counterpart, the following trick should help you with this*

```
MyModel.objects.filter(...).sql_with_params()
```

- *Create SQL functions and use them with the* `Func` *object*
- *Use manager methods with meticulously crafted raw sql and use those methods in your code.*

So yes, use your ORM. Not the one from Django. Yours !

What do you think of supporting several RDMS solutions in your applications?

Sorry but I have to admit that back in the days I believed in such a tale. Now as a grown-up I know two things. Santa and RDBMS agnosticism do not really exist.

What is true is that a framework like Django lets you choose a database and then stick with it.

The idea of using SQLite in development and PostgreSQL in production leads only to one thing: you will use the features of SQLite everywhere and you will not be able to use the PostgreSQL specific features.

The only way to be purely agnostic is to use only the features all the proposed RDMS provides. But think again. Do you want to drive your race car like a tractor?

4

SQL Toolbox

Contents

In this chapter, we are going to add to our proficiency in writing SQL queries. The *structured query language* doesn't look like any other imperative, functional or even object-oriented programming language.

This chapter contains a long list of SQL techniques from the most basic *select* clause to advanced *lateral joins*, each time with practical examples working with a free database that you can install at home.

It is highly recommended that you follow along with a local instance of the database so that you can enter the queries from the book and play with them yourself. A key aspect of this part is that SQL queries arent' typically written in a text editor with hard thinking, instead they are interactively tried out in pieces and stitched together once the spelling is spot on.

The SQL writing process is mainly about discovery. In SQL you need to explain your problem, unlike in most programming languages where you need to focus on a solution you think is going to solve your problem. That's quite different and requires looking at your problem in another way and understanding it well enough to be able to express it in details in a single sentence.

Here's some good advice I received years and years ago, and it still applies to this day: when you're struggling to write a SQL query, first write down a single sentence —in your native language— that perfectly describes what you're trying to achieve. As soon as you can do that, then writing the SQL is going to be easier.

One of the very effective techniques in writing such a sentence is talking out loud, because apparently writing and speaking come from different parts of the brain. So it's the same as when debugging a complex program, as it helps a lot to talk about it with a colleague... or a rubber duck.

After having dealt with the basics of the language, where means basic really **fundamentals**, this chapter spends time on more advanced SQL concepts and PostgreSQL along with how you can benefit from them when writing your applications, making you a more effective developer.

Get Some Data

To be able to play with SQL queries, we first need some data. While it is possible to create a synthetic set of data and play with it, it is usually harder to think about abstract numbers that you know nothing about.

In this chapter, we are going to use the historical record of motor racing data, available publicly.

The database is available in a single download file for MySQL only. Once

you have a local copy, we use pgloader to get the data set in PostgreSQL:

```
$ createdb f1db
$ pgloader mysql://root@localhost/f1db pgsql:///f1db
```

Now that we have a real data set, we can get into more details about the window function frames. To run the query as written in the following parts, you also need to tweak PostgreSQL *search_path* to include the *f1db* schema in the *f1db* database. Here's the SQL command you need for that:

```
ALTER DATABASE f1db SET search_path TO f1db, public;
```

When using the *Full Edition* or the *Enterprise Edition* of the book, the `appdev` database is already loaded with the dataset in the `f1db` schema.

Structured Query Language

SQL stands for *structured query language* and has been designed so that non-programmer would be able to use it for their reporting needs. Ignoring this clear attempt at getting Marketing people to stay away from the developer's desks, this explains why the language doesn't look like your normal programming language.

Apart from the aim to look like English sentences, the main aspect of the SQL language to notice and learn to benefit from is that it's a *declarative* programming language. This means that you get to *declare* or *state* the result you want to obtain, thus you need to think in term of the problem you want to solve.

This differs from most programming languages, where the developer's job is to transform his understanding of the solution into a step by step recipe for how exactly to obtain it, which means thinking in terms of the solution you decided would solve the problem at hand.

It is then quite fair to say that SQL is a very high-level programming language: even as a developer you don't need to come up with a detailed solution, rather your job is to understand the problem well enough so that you are able to translate it. After that, the RDBMS of your choice is going to figure out a plan then execute it, and hopefully return just the result set you wanted!

For some developers, not being in charge of every detail of the query plan is a source of frustration, and they prefer hiding SQL under another layer of technology that makes them feel like they are still in control.

Unfortunately, any extra layer on top of SQL is only there to produce SQL for you, which means you have even less control over what plan is going to be executed.

In this section, we review important and basic parts of a SQL query. The goal is for you to be comfortable enough with writing SQL that you don't feel like you've lost control over the details of its execution plan, but instead your can rely on your RDBMS of choice for that. Of course, it's much easier to reach that level of trust when you use PostgreSQL, because it is fully open source, well documented, supports a very detailed *explain* command, and its code is very well commented, making it easy enough to read and review.

Queries, DML, DDL, TCL, DCL

SQL means *structured query language* and is composed of several areas, and each of them has a specific acronym and sub-language.

- *DML* stands for *data manipulation language* and it covers *insert*, *update* and *delete* statements, which are used to input data into the system.
- *DDL* stands for *data definition language* and it covers *create*, *alter* and *drop* statements, which are used to define on-disk data structures where to hold the data, and also their constraints and indexes — the things we refer to with the terms of *SQL objects*.
- *TCL* stands for *transaction control language* and includes *begin* and *commit* statements, and also *rollback*, *start transaction* and *set transaction* commands. It also includes the less well-known *savepoint*, *release savepoint*, and *rollback to savepoint* commands, and let's not forget about the two-phase commit protocol with *prepare commit*, *commit prepared* and *rollback prepared* commands.
- *DCL* stands for *data control language* and is covered with the state-

ments *grant* and *revoke*.

- Next we have PostgreSQL maintenance commands such as *vacuum*, *analyze*, *cluster*.
- There further commands that are provided by PostgreSQL such as *prepare* and *execute*, *explain*, *listen* and *notify*, *lock* and *set*, and some more.

The *query* part of the language, which covers statements beginning with *select*, *table*, *values* and *with* keywords, is a tiny part of the available list of commands. It's also where the complexity lies and the part we are going to focus our efforts in this section.

Select, From, Where

Anatomy of a Select Statement

The simplest *select* statement in PostgreSQL is the following:

```
1 | SELECT 1;
```

In other systems, the *from* clause is required and sometimes a dummy table with a single row is provided so that you can *select* from this table.

Projection (output): Select

The SQL *select* clause introduces the list of output columns. This is the list of data that we are going to send back to the client application, so it's quite important: the only reason the server is executing any query is to return a result set where each row presents the list of columns specified in the *select* clause. This is called a *projection*.

Adding a column to the *select* list might involve a lot of work, such as:

- Fetching data on-disk
- Possibly uncompressing data that is stored externally to the main table on-disk structure, and loading those uncompressed bytes into the memory of the database server

- Sending the data back over the network back to the client application.

Given that, it is usually frowned upon to use either the infamous *select star* notation or the classic *I don't know what I'm doing* behavior of some object relational mappers when they insist on always fully *hydrating* the application objects, just in case.

The following shortcut is nice to have in interactive mode only:

```
select * from races limit 1;
```

The actual standard syntax for *limit* is a little more complex:

```
select * from races fetch first 1 rows only;
```

It gives the following result:

```
-[ RECORD 1 ]-------------------------------------------------------
raceid    | 1
year      | 2009
round     | 1
circuitid | 1
name      | Australian Grand Prix
date      | 2009-03-29
time      | 06:00:00
url       | http://en.wikipedia.org/wiki/2009_Australian_Grand_Prix
```

Note that rather than using this frowned upon notation, the SQL standard allows us to use this alternative, which is even more practical:

```
table races limit 1;
```

Of course, it gives the same result as the one above.

Select Star

There's another reason to refrain from using the *select star* notation in application's code: if you ever change the source relation definitions, then the same query now has a different result set data structure, and you might have to reflect that change in the application's in-memory data structures.

Let's take a very simple Java example, and I will only show the meat of it, filtering out the exception handling and resources disposal (we need to close the result set, the statement and the connection objects):

```
try {
  con = DriverManager.getConnection(url, user, password);
```

```
  st = con.createStatement();
  rs = st.executeQuery("SELECT * FROM races LIMIT 1;");

  if (rs.next()) {
      System.out.println(rs.getInt("raceid"));
      System.out.println(rs.getInt("year"));
      System.out.println(rs.getInt("round"));
      System.out.println(rs.getInt("circuitid"));
      System.out.println(rs.getString("name"));
      System.out.println(rs.getString("date"));
      System.out.println(rs.getString("time"));
      System.out.println(rs.getString("url"));
  }
} catch (SQLException ex) {
  // logger code
} finally {
  // closing code
}
```

We can use the file like this:

```
$ javac Select.java
$ java -cp .:path/to/postgresql-42.1.1.jar Select
1
2009
1
1
Australian Grand Prix
2009-03-29
06:00:00
http://en.wikipedia.org/wiki/2009_Australian_Grand_Prix
```

Even in this pretty quick example we can see that the code has to know the *races* table column list, each column name, and the data types. Of course, it's still possible to write the following code:

```
if (rs.next()) {
    for(int i=1; i<=8; i++)
        System.out.println(rs.getString(i));
}
```

But this case is only relevant when we have no processing at all to do over the data, and we still hard code the fact that the *races* table has eight column.

Now pretend we had an *extra* column in our schema definition at some point, and thus had the following line in our code to process it from the result set:

```
      System.out.println(rs.getString("extra"));
```

Once the column is no longer here (presumably following a production rollout of the schema change), then our code no longer runs:

```
Jun 29, 2017 1:17:41 PM Select main
SEVERE: The column name extra was not found in this ResultSet.
org.postgresql.util.PSQLException: The column name extra was not found in this ResultS
    at org.postgresql.jdbc.PgResultSet.findColumn(PgResultSet.java:2610)
    at org.postgresql.jdbc.PgResultSet.getString(PgResultSet.java:2484)
    at Select.main(Select.java:35)
```

That's because now our code is wrong, and code review can't help us here, because the query in both cases is a plain `select * ...`. We could have used the following code instead:

```
try {
    con = DriverManager.getConnection(url, user, password);
    st = con.createStatement();
    rs = st.executeQuery("SELECT name, date, url, extra FROM races LIMIT 1;");

    if (rs.next()) {
        System.out.println(" race: " + rs.getString("name"));
        System.out.println(" date: " + rs.getString("date"));
        System.out.println("  url: " + rs.getString("url"));
        System.out.println("extra: " + rs.getString("url"));
        System.out.println();
    }

} catch (SQLException ex) {
  // logger code
} finally {
  // closing code
}
```

Now it's quite clear that there's a direct mapping between the column names in the SQL query and what we fetch from the result set instance. We still don't know at review or compile time if the columns do currently exist in production, but at least the error message is crystal clear this time:

```
Jun 29, 2017 1:31:04 PM Select main
SEVERE: ERROR: column "extra" does not exist
  Position: 25
org.postgresql.util.PSQLException: ERROR: column "extra" does not exist
```

Again, when being explicit, the *diff* is pretty easy to review too:

```
@@ -21,18 +21,17 @@
 try {
     con = DriverManager.getConnection(url, user, password);
     st = con.createStatement();
-    rs = st.executeQuery("SELECT name, date, url, extra FROM races LIMIT 1;");
+    rs = st.executeQuery("SELECT name, date, url FROM races LIMIT 1;");
```

```
7
8       if (rs.next()) {
9           System.out.println(" race: " + rs.getString("name"));
10          System.out.println(" date: " + rs.getString("date"));
11          System.out.println("  url: " + rs.getString("url"));
12  -       System.out.println("extra: " + rs.getString("extra"));
13          System.out.println();
14      }
```

To summarize, here's a review of my argument against *select star*:

- Using `select *` hides the intention of the code, while listing the columns explicitly in the code allows for declaring our thinking as a developer.
- It makes code changes easier to review when the column list is explicit in the code, and despite our previous example in Java using a string literal as a SQL query, it's even better of course when the query is found in a proper `.sql` file.
- It is not efficient to retrieve all the bytes each time even if you don't need them, some bytes are quite expensive to fetch on the server side thanks to the TOAST mechanism (The Oversized-Attribute Storage Technique), and then those bytes still need to find their way in the network and your application's memory.

The main point is about being specific about what your code is doing. It helps tremendously to never have to second guess what is happening, for example in cases of production debugging, performances analysis and optimization, onboarding of new team members, code review, and really just about anything that has to do with maintaining the code base.

Select Computed Values and Aliases

In the *SELECT* clause it is possible to return computed values and to rename columns. Here's an example of that:

```
1  select code,
2         format('%s %s', forename, surname) as fullname,
3         forename,
4         surname as fullname
5   from drivers;
```

And here are the first three drivers we get:

```
 code │     fullname     │ forename │ fullname
══════╪══════════════════╪══════════╪══════════
 HAM  │ Lewis Hamilton   │ Lewis    │ Hamilton
 HEI  │ Nick Heidfeld    │ Nick     │ Heidfeld
 ROS  │ Nico Rosberg     │ Nico     │ Rosberg
```

Here we are using the `format` PostgreSQL function, which mimics what is usually available in programming languages such as Python's *print* function or C's *printf*. The SQL standard gives us a concatenation operator named `||` and we could achieve the same result with a standard conforming query:

```
select code,
       forename || ' ' || surname as fullname,
       forename,
       surname as fullname
 from drivers;
```

In this book, we are going to focus on PostgreSQL rather than standard compliance, because PostgreSQL offers a lot of useful functions and gems that are nowhere to be found in the SQL standard, nor in most of the RDBMS competition.

The visibility of the *SELECT* alias is important to keep in mind. This is a topic for later in this chapter, when we learn about the *ORDER BY*, *GROUP BY*, *HAVING* and *WINDOW* clauses.

PostgreSQL Processing Functions

PostgreSQL embeds a very rich set of processing functions that can be used anywhere in the queries, even if most of them are more useful in the *SELECT* clause. Because I see a lot of code fetching only the raw data from the RDBMS and then doing all the processing in the application code, I want to show an example query processing calendar related information with PostgreSQL.

The next query is a showcase for *extract()* and *to_char()* functions, and it also uses the *CASE* construct. Read the documentation on date/time functions and operators for more details and functions on the same topic.

```
select date::date,
       extract('isodow' from date) as dow,
       to_char(date, 'dy') as day,
       extract('isoyear' from date) as "iso year",
```

```
5         extract('week' from date) as week,
6         extract('day' from
7                 (date + interval '2 month - 1 day')
8                )
9          as feb,
10        extract('year' from date) as year,
11        extract('day' from
12                (date + interval '2 month - 1 day')
13               ) = 29
14        as leap
15    from generate_series(date '2000-01-01',
16                         date '2010-01-01',
17                         interval '1 year')
18        as t(date);
```

The *generate_series()* function returns a set of items, here all the dates of the first day of the years from the 2000s. For each of them we then compute the day of the week of this first day of the year, both in numerical and textual forms, and then the year number from the date, as defined by the ISO standard, and the week number from the ISO year, then the last day of February and a Boolean which is true for leap years.

Here's an extract from the PostgreSQL documentation about ISO years and week numbers:

> By definition, ISO weeks start on Mondays and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year.

So here's what we get:

```
    date    | dow | day | iso year | week | feb | year | leap
============+=====+=====+==========+======+=====+======+======
 2000-01-01 |   6 | sat |     1999 |   52 |  29 | 2000 | t
 2001-01-01 |   1 | mon |     2001 |    1 |  28 | 2001 | f
 2002-01-01 |   2 | tue |     2002 |    1 |  28 | 2002 | f
 2003-01-01 |   3 | wed |     2003 |    1 |  28 | 2003 | f
 2004-01-01 |   4 | thu |     2004 |    1 |  29 | 2004 | t
 2005-01-01 |   6 | sat |     2004 |   53 |  28 | 2005 | f
 2006-01-01 |   7 | sun |     2005 |   52 |  28 | 2006 | f
 2007-01-01 |   1 | mon |     2007 |    1 |  28 | 2007 | f
 2008-01-01 |   2 | tue |     2008 |    1 |  29 | 2008 | t
 2009-01-01 |   4 | thu |     2009 |    1 |  28 | 2009 | f
 2010-01-01 |   5 | fri |     2009 |   53 |  28 | 2010 | f
(11 rows)
```

It is very easy to do complex computations on dates in PostgreSQL, and that includes taking care of time zones too. Don't even think about coding

such processing yourself, as it's full of oddities.

Data sources: From

The SQL *from* clause introduces the data sources used in the query, and supports declaring how those different sources relate to each other. In the most basic form, our query is reading a data set from a single table:

```
select code, driverref, forename, surname
  from drivers;
```

In this query *drivers* is the name of a table, so it's pretty easy to understand what's going on.

Now say we want to get the all-time top three drivers in terms of how many times they won a race. This time we need information from the *drivers* table and from the *results* table, which along with other information contains a *position* column. The winner's position is 1.

To find the all-time top three drivers, we fetch how many times each driver had *position = 1* in the results table:

```
  select code, forename, surname,
         count(*) as wins
    from      drivers
         join results using(driverid)
   where position = 1
group by driverid
order by wins desc
limit 3;
```

This time the result is more interesting. let's have a look at our all time top three winners in the Formula One database:

```
 code │ forename │  surname   │ wins
══════╪══════════╪════════════╪══════
 MSC  │ Michael  │ Schumacher │   91
 HAM  │ Lewis    │ Hamilton   │   56
 ¤    │ Alain    │ Prost      │   51
(3 rows)
```

The query uses an *inner join* in between the *drivers* and the *results* table. In both those tables, there is a *driverid* column that we can use as a lookup reference to associate data in between the two tables.

Understanding Joins

I could spend time here and fill in the book with detailed explanations of every kind of *join* operation: *inner join*, *left* and *right outer joins*, *cross joins*, *full outer join*, *lateral join* and more. It just so happens that the PostgreSQL documentation covering the FROM clause does that very well, so please read it carefully along with this book so that we can instead focus on more interesting and advanced examples.

Now that we know how to easily fetch the winner of a race, it is possible to also to display all the races from a quarter with their winner:

```
\set beginning '2017-04-01'
\set months 3

select date, name, drivers.surname as winner
  from races
       left join results
              on results.raceid = races.raceid
             and results.position = 1
       left join drivers using(driverid)
 where date >= date :'beginning'
   and date <  date :'beginning'
              + :months * interval '1 month';
```

And we get the following result, where we lack data for the most recent race but still display it:

```
    date    │          name          │  winner
════════════╪════════════════════════╪══════════
 2017-04-09 │ Chinese Grand Prix     │ Hamilton
 2017-04-16 │ Bahrain Grand Prix     │ Vettel
 2017-04-30 │ Russian Grand Prix     │ Bottas
 2017-05-14 │ Spanish Grand Prix     │ Hamilton
 2017-05-28 │ Monaco Grand Prix      │ Vettel
 2017-06-11 │ Canadian Grand Prix    │ Hamilton
 2017-06-25 │ Azerbaijan Grand Prix  │ ¤
(7 rows)
```

The reason why we are using a *left join* this time is so that we keep every race from the quarter's and display extra information only when we have it. *Left join* semantics are to keep the whole result set of the table lexically on the left of the operator, and to fill-in the columns for the table on the right of the *left join* operator when some data is found that matches the *join condition*, otherwise using NULL as the column's value.

In the example above, the *winner* information comes from the *results* table,

which is lexically found at the right of the *left join* operator. The *Azerbaijan Grand Prix* has no results in the local copy of the *f1db* database used locally, so the *winner* information doesn't exists and the SQL query returns a *NULL* entry.

You can also see that the *results.position = 1* restriction has been moved directly into the join condition, rather than being kept in the where clause. Should the condition be in the *where* clause, it would filter out races from which we don't have a result yet, and we are still interested into those.

Another way to write the query would be using an explicit subquery to build an intermediate results table containing only the winners, and then join against that:

```
select date, name, drivers.surname as winner
  from races
       left join
           ( select raceid, driverid
               from results
              where position = 1
           )
           as winners using(raceid)
       left join drivers using(driverid)
 where date >= date :'beginning'
   and date <  date :'beginning'
              + :months * interval '1 month';
```

PostgreSQL is smart enough to actually implement both SQL queries the same way, but it might be thanks to the data set being very small in the f1db database.

Restrictions: Where

In most of the queries we saw, we already had some *where* clause. This clause acts as a filter for the query: when the filter evaluates to true then we keep the row in the result set and when the filter evaluates to false we skip that row.

Real-world SQL may have quite complex *where* clauses to deal with, and it is allowed to use *CASE* and other logic statements. That said, we usually try to keep the *where* clauses as simple as possible for PostgreSQL in order to be able to use our indexes to solve the data filtering expressions of our queries.

Some simple rules to remember here:

- In a *where* clause we can combine filters, and generally we combine them with the *and* operator, which allows short-circuit evaluations because as soon as one of the *anded* conditions evaluates to false, we know for sure we can skip the current row.
- *Where* also supports the *or* operator, which is more complex to optimize for, in particular with respect to indexes.
- We have support for both *not* and *not in*, which are completely different beasts.

  Be careful about *not in* semantics with *NULL*: the following query returns no rows...

```
select x
  from generate_series(1, 100) as t(x)
 where x not in (1, 2, 3, null);
```

Finally, as is the case just about anywhere else in a SQL query, it is possible in the *where* clause to use a subquery, and that's quite common to use when implementing the *anti-join* pattern thanks to the special feature *not exists*.

An *anti-join* is meant to keep only the rows that fail a test. If we want to list the drivers that where unlucky enough to not finish a single race in which they participated, then we can filter out those who did finish. We know that a driver finished because their *position* is filled in the *results* table: it *is not null*.

If we translate the previous sentence into the SQL language, here's what we have:

```
\set season 'date ''1978-01-01'''

  select forename,
         surname,
         constructors.name as constructor,
         count(*) as races,
         count(distinct status) as reasons

    from drivers
         join results using(driverid)
         join races using(raceid)
         join status using(statusid)
         join constructors using(constructorid)

   where date >= :season
```

```
16      and date <  :season + interval '1 year'
17      and not exists
18          (
19            select 1
20              from results r
21             where position is not null
22               and r.driverid = drivers.driverid
23               and r.resultid = results.resultid
24          )
25  group by constructors.name, driverid
26  order by count(*) desc;
```

The interesting part of this query lies in the *where not exists* clause, which might look somewhat special on a first read: what is that *select 1* doing there?

Remember that a *where* clause is a filter. The *not exists* clause is filtering based on rows that are returned by the subquery. To pass the filter, just return anything, PostgreSQL will not even look at what is selected in the subquery, it will only take into account the fact that a row was returned.

It also means that the join condition in between the main query and the *not exists* subquery is done in the *where* clause of the subquery, where you can reference the outer query as we did in *r.driverid = drivers.driverid and r.resultid = results.resultid*.

It turns out that 1978 was not a very good season based on the number of drivers who never got the chance to finish a race so we are going to show only the ten first results of the query:

```
   forename    │  surname  │ constructor │ races │ reasons
═══════════════╪═══════════╪═════════════╪═══════╪═════════
 Arturo        │ Merzario  │ Merzario    │    16 │       8
 Hans-Joachim  │ Stuck     │ Shadow      │    12 │       6
 Rupert        │ Keegan    │ Surtees     │    12 │       6
 Hector        │ Rebaque   │ Team Lotus  │    12 │       7
 Jean-Pierre   │ Jabouille │ Renault     │    10 │       4
 Clay          │ Regazzoni │ Shadow      │    10 │       5
 James         │ Hunt      │ McLaren     │    10 │       6
 Brett         │ Lunger    │ McLaren     │     9 │       5
 Niki          │ Lauda     │ Brabham     │     9 │       4
 Rolf          │ Stommelen │ Arrows      │     8 │       5
(10 rows)
```

The reasons not to finish a race might be *did not qualify* or *gearbox*, or any one of the 133 different statuses found in the f1db database.

Order By, Limit, No Offset

Ordering with Order By

The SQL *ORDER BY* clause is pretty well-known because SQL doesn't guarantee any ordering of the result set of any query except when you use the *order by* clause.

In its simplest form the *order by* works with one column or several columns that are part of our data model, and in some cases, it might even allow PostgreSQL to return the data in the right order by following an existing index.

```
  select year, url
    from seasons
order by year desc
   limit 3;
```

This gives an expected and not that interesting result set:

```
 year │                          url
══════╪═══════════════════════════════════════════════════
 2017 │ https://en.wikipedia.org/wiki/2017_Formula_One_season
 2016 │ https://en.wikipedia.org/wiki/2016_Formula_One_season
 2015 │ http://en.wikipedia.org/wiki/2015_Formula_One_season
(3 rows)
```

What is more interesting about it is the *explain plan* of the query, where we see PostgreSQL follows the primary key index of the table in a backward direction in order to return our three most recent entries. We obtain the plan with the following query:

```
 explain (costs off)
  select year, url
    from seasons
order by year desc
   limit 3;
```

Well, this one is pretty easy to read and understand:

```
                        QUERY PLAN
═══════════════════════════════════════════════════════════
 Limit
   ->  Index Scan Backward using idx_57708_primary on seasons
(2 rows)
```

The *order by* clause can also refer to query aliases and computed values, as

we noted earlier in previous queries. More complex use cases are possible: in PostgreSQL, the clause also accepts complex expression and subqueries.

As an example of a complex expression, we may use the *CASE* conditional in order to control the ordering of a race's results over the status information. Say that we order the results by position then number of laps and then by status with a special rule: the *Power Unit* failure condition is considered first, and only then the other ones.

Yes, this rule makes no sense at all, it's totally arbitrary. It could be that you're working with a constructor and he's making a study about some failing hardware and that's part of the inquiry.

```
select drivers.code, drivers.surname,
       position,
       laps,
       status
  from results
       join drivers using(driverid)
       join status using(statusid)
 where raceid = 972
order by position nulls last,
         laps desc,
         case when status = 'Power Unit'
              then 1
              else 2
          end;
```

We can almost feel we've seen the race with that result set:

code	surname	position	laps	status
BOT	Bottas	1	52	Finished
VET	Vettel	2	52	Finished
RAI	Räikkönen	3	52	Finished
HAM	Hamilton	4	52	Finished
VER	Verstappen	5	52	Finished
PER	Pérez	6	52	Finished
OCO	Ocon	7	52	Finished
HUL	Hülkenberg	8	52	Finished
MAS	Massa	9	51	+1 Lap
SAI	Sainz	10	51	+1 Lap
STR	Stroll	11	51	+1 Lap
KVY	Kvyat	12	51	+1 Lap
MAG	Magnussen	13	51	+1 Lap
VAN	Vandoorne	14	51	+1 Lap
ERI	Ericsson	15	51	+1 Lap
WEH	Wehrlein	16	50	+2 Laps
RIC	Ricciardo	¤	5	Brakes

```
 ALO  | Alonso     |        ¤ |    0 | Power Unit
 PAL  | Palmer     |        ¤ |    0 | Collision
 GRO  | Grosjean   |        ¤ |    0 | Collision
(20 rows)
```

kNN Ordering and GiST indexes

Another use case for *order by* is to implement *k nearest neighbours*. The *kNN* searches are pretty well covered in the literature and is easy to implement in PostgreSQL. Let's find out the ten nearest circuits to Paris, France, which is at longitude 2.349014 and latitude 48.864716. That's a kNN search with k = 10:

```
  select name, location, country
    from circuits
order by point(lng,lat) <-> point(2.349014, 48.864716)
   limit 10;
```

Along with the following list of circuits spread around in France, we also get some tracks from Belgium and the United Kingdom:

```
            name             |     location      | country
═════════════════════════════╪═══════════════════╪═════════
 Rouen-Les-Essarts           | Rouen             | France
 Reims-Gueux                 | Reims             | France
 Circuit de Nevers Magny-Cours | Magny Cours     | France
 Le Mans                     | Le Mans           | France
 Nivelles-Baulers            | Brussels          | Belgium
 Dijon-Prenois               | Dijon             | France
 Charade Circuit             | Clermont-Ferrand  | France
 Brands Hatch                | Kent              | UK
 Zolder                      | Heusden-Zolder    | Belgium
 Circuit de Spa-Francorchamps | Spa              | Belgium
(10 rows)
```

The *point* datatype is a very useful PostgreSQL addition. In our query here, the points have been computed from the raw data in the database. For a proper PostgreSQL experience, we can have a location column of point type in our circuits table and index it using GiST:

```
begin;

alter table f1db.circuits add column position point;
update f1db.circuits set position = point(lng,lat);
create index on f1db.circuits using gist(position);

commit;
```

Now the previous query can be written using the new column. We get the same result set, of course: indexes are not allowed to change the result of a query they apply to... under no circumstances. When they do, we call that a bug, or maybe it is due to data corruption. Anyway, let's have a look at the query plan now that we have a *GiST* index defined:

```
explain (costs off, buffers, analyze)
  select name, location, country
    from circuits
order by position <-> point(2.349014, 48.864716)
   limit 10;
```

The *(costs off)* option is used here mainly so that the output of the command fits in the book's page format, so try without the option at home:

```
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
 Limit (actual time=0.039..0.061 rows=10 loops=1)
   Buffers: shared hit=7
   ->  Index Scan using circuits_position_idx on circuits
        (actual time=0.038..0.058 rows=10 loops=1)
         Order By: ("position" <-> '(2.349014,48.864716)'::point)
         Buffers: shared hit=7
 Planning time: 0.129 ms
 Execution time: 0.105 ms
(7 rows)
```

We can see that PostgreSQL is happy to be using our GiST index and even goes so far as to implement our whole kNN search query all within the index. For reference the query plan of the previous spelling of the query, the dynamic expression *point(lng,lat)* looks like this:

```
explain (costs off, buffers, analyze)
  select name, location, country
    from circuits
order by point(lng,lat) <-> point(2.349014, 48.864716)
   limit 10;
```

And here's the query plan when not using the index:

```
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
 Limit (actual time=0.246..0.256 rows=10 loops=1)
   Buffers: shared hit=5
   ->  Sort (actual time=0.244..0.249 rows=10 loops=1)
         Sort Key: ((point(lng, lat) <-> '(2.349014,48.864716)'::point))
         Sort Method: top-N heapsort  Memory: 25kB
         Buffers: shared hit=5
         ->  Seq Scan on circuits
              (actual time=0.024..0.133 rows=73 loops=1)
               Buffers: shared hit=5
 Planning time: 0.189 ms
 Execution time: 0.344 ms
```

```
(10 rows)
```

By default, the distance operator `<->` is defined only for geometric data types in PostgreSQL. Some extensions such as pg_trgm add to that list so that you may benefit from a kNN index lookup in other situations, such as in queries using the *like* operator, or even the regular expression operator `~`. You'll find more on regular expressions in PostgreSQL later in this book.

Top-N sorts: Limit

It would be pretty interesting to get the list of the **top three drivers** in terms of races won, **by decade**. It is possible to do so thanks to advanced PostgreSQL date functions manipulation together with implementation of lateral joins.

The following query is a classic top-N implementation. It reports for each decade the top three drivers in terms of race wins. It is both a classic top-N because it is done thanks to a *lateral* subquery, and at the same time it's not so classic because we are joining against computed data. The decade information is not part of our data model, and we need to extract it from the *races.date* column.

```
with decades as
(
   select extract('year' from date_trunc('decade', date)) as decade
     from races
 group by decade
)
select decade,
       rank() over(partition by decade
                   order by wins desc)
       as rank,
       forename, surname, wins

  from decades
       left join lateral
       (
          select code, forename, surname, count(*) as wins
            from drivers

                 join results
                   on results.driverid = drivers.driverid
                  and results.position = 1

                 join races using(raceid)
```

```
24
25            where   extract('year' from date_trunc('decade', races.date))
26                  = decades.decade
27
28         group by decades.decade, drivers.driverid
29         order by wins desc
30            limit 3
31        )
32        as winners on true
33
34 order by decade asc, wins desc;
```

The query extracts the decade first, in a *common table expression* introduced with the *with* keyword. This *CTE* is then reused as a data source in the *from* clause. The *from* clause is about relations, which might be hosting a dynamically computed dataset, as is the case in this example.

Once we have our list of decades from the dataset, we can fetch for each decade the list of the top three winners for each decade from the *results* table. The best way to do that in SQL is using a *lateral* join. This form of join allows one to write a subquery that runs in a loop over a data set. Here we loop over the decades and for each decade our *lateral subquery* finds the top three winners.

Focusing now on the *winners* subquery, we want to *count* how many times a driver made it to the first position in a race. As we are only interested in winning results, the query pushes that restriction in the *join condition* of the *left join results* part. The subquery should also only count victories that happened in the current decade from our loop, and that's implemented in the *where* clause, because that's how *lateral* subqueries work. Another interesting implication of using a *left join lateral subquery* is how the join clause is then written: *on true*. That's because we inject the join condition right into the subquery as a where clause. This trick allows us to only see the results from the current decade in the subquery, which then uses a *limit* clause on top of the *order by wins desc* to report the top three with the most wins.

And here's the result of our query:

decade	rank	forename	surname	wins
1950	1	Juan	Fangio	24
1950	2	Alberto	Ascari	13
1950	3	Stirling	Moss	12
1960	1	Jim	Clark	25

```
   1960 |    2 | Graham    | Hill       |   14
   1960 |    3 | Jack      | Brabham    |   11
   1970 |    1 | Niki      | Lauda      |   17
   1970 |    2 | Jackie    | Stewart    |   16
   1970 |    3 | Emerson   | Fittipaldi |   14
   1980 |    1 | Alain     | Prost      |   39
   1980 |    2 | Nelson    | Piquet     |   20
   1980 |    2 | Ayrton    | Senna      |   20
   1990 |    1 | Michael   | Schumacher |   35
   1990 |    2 | Damon     | Hill       |   22
   1990 |    3 | Ayrton    | Senna      |   21
   2000 |    1 | Michael   | Schumacher |   56
   2000 |    2 | Fernando  | Alonso     |   21
   2000 |    3 | Kimi      | Räikkönen  |   18
   2010 |    1 | Lewis     | Hamilton   |   45
   2010 |    2 | Sebastian | Vettel     |   40
   2010 |    3 | Nico      | Rosberg    |   23
(21 rows)
```

No Offset, and how to implement pagination

The SQL standard offers a *fetch* command instead of the *limit* and *offset* variant that we have in PostgreSQL. In any case, using the *offset* clause is very bad for your query performances, so we advise against it:

Figure 4.1: No Offset

Please take the time to read Markus Winand's Paging Through Results, as I won't explain it better than he does. Also, never use *offset* again!

As easy as it is to task you to read another article online, and as good as it is, it still seems fair to give you the main take away in this book's pages. The *offset* clause is going to cause your SQL query plan to read all the result anyway and then discard most of it until reaching the *offset* count. When paging through lots of results, it's less and less efficient with each additional page you fetch that way.

The proper way to implement pagination is to use index lookups, and if you have multiple columns in your ordering clause, you can do that with the *row()* construct.

To show an example of the method, we are going to paginate through the *laptimes* table, which contains every lap time for every driver in any race. For the raceid 972 that we were having a look at earlier, that's a result with 828 lines. Of course, we're going to need to paginate through it.

Here's how to do it properly, given pages of three rows at a time, to save space in this book for more interesting text. The first query is as expected:

```
    select lap, drivers.code, position,
           milliseconds * interval '1ms' as laptime
      from laptimes
           join drivers using(driverid)
     where raceid = 972
  order by lap, position
fetch first 3 rows only;
```

We are using the SQL standard spelling of the *limit* clause here, and we get the first page of lap timings for the race:

```
 lap | code | position |       laptime
═════╪══════╪══════════╪═══════════════════
   1 | BOT  |        1 | @ 2 mins 5.192 secs
   1 | VET  |        2 | @ 2 mins 7.101 secs
   1 | RAI  |        3 | @ 2 mins 10.53 secs
(3 rows)
```

The result set is important because your application needs to make an effort here and remember that it did show you the results up until *lap = 1* and *position = 3*. We are going to use that so that our next query shows the next page of results:

```
    select lap, drivers.code, position,
           milliseconds * interval '1ms' as laptime
```

```
3 |       from laptimes
4 |            join drivers using(driverid)
5 |      where raceid = 972
6 |        and row(lap, position) > (1, 3)
7 |   order by lap, position
8 | fetch first 3 rows only;
```

And here's our second page of query results. After a first page finishing at lap 1, position 3 we are happy to find out a new page beginning at lap 1, position 4:

```
 lap | code | position |       laptime
═════╪══════╪══════════╪═══════════════════════
   1 | HAM  |        4 | @ 2 mins 11.18 secs
   1 | VER  |        5 | @ 2 mins 12.202 secs
   1 | MAS  |        6 | @ 2 mins 13.501 secs
(3 rows)
```

So please, never use *offset* again if you care at all about your query time!

Group By, Having, With, Union All

Now that we have some of the basics of SQL queries, we can move on to more advanced topics. Up to now, queries would return as many rows as we select thanks to the *where* filtering. This filter applies against a data set that is produced by the *from* clause and its *joins* in between relations.

The *outer* joins might produce more rows than you have in your reference data set, in particular, *cross join* is a Cartesian product.

In this section, we'll have a look at **aggregates**. They work by computing a digest value for several input rows at a time. With aggregates, we can return a summary containing many fewer rows than passed the *where* filter.

Aggregates (aka Map/Reduce): Group By

The *group by* clause introduces aggregates in SQL, and allows implementing much the same thing as *map/reduce* in other systems: map your data into different groups, and in each group reduce the data set to a single value.

As a first example we can count how many races have been run in each decade:

```
select extract('year'
               from
               date_trunc('decade', date))
       as decade,
       count(*)
  from races
group by decade
order by decade;
```

PostgreSQL offers a rich set of date and times functions:

```
 decade | count
════════╪═══════
   1950 |    84
   1960 |   100
   1970 |   144
   1980 |   156
   1990 |   162
   2000 |   174
   2010 |   156
(7 rows)
```

The difference between each decade is easy to compute thanks to *window function*, seen later in this chapter. Let's have a preview:

```
with races_per_decade
as (
      select extract('year'
                     from
                     date_trunc('decade', date))
             as decade,
             count(*) as nbraces
        from races
      group by decade
      order by decade
   )
select decade, nbraces,
       case
         when lag(nbraces, 1)
              over(order by decade) is null
         then ''

         when nbraces - lag(nbraces, 1)
                        over(order by decade)
              < 0
         then format('-%3s',
                lag(nbraces, 1)
                over(order by decade)
                - nbraces)
```

```
25
26          else format('+%3s',
27                 nbraces
28               - lag(nbraces, 1)
29                over(order by decade))
30
31        end as evolution
32   from races_per_decade;
```

We use a pretty complex *CASE* statement to elaborate on the exact output we want from the query. Other than that it's using the *lag() over(order by decade)* expression that allows seeing the previous row, and moreover allows us to compute the difference in between the current row and the previous one.

Here's what we get from the previous query:

```
 decade | nbraces | evolution
════════╪═════════╪═══════════
   1950 |      84 |
   1960 |     100 | + 16
   1970 |     144 | + 44
   1980 |     156 | + 12
   1990 |     162 | +  6
   2000 |     174 | + 12
   2010 |     156 | - 18
(7 rows)
```

Now, we can also prepare the data set in a separate query that is run first, called a *common table expression* and introduced by the *with* clause. We will expand on that idea in the upcoming pages.

PostgreSQL comes with the usual aggregates you would expect such as *sum*, *count*, and *avg*, and also with some more interesting ones such as *bool_and*. As its name suggests the *bool_and* aggregate starts true and remains true only if every row it sees evaluates to true.

With that aggregate, it's then possible to search for all drivers who failed to finish any single race they participated in over their whole career:

```
1  with counts as
2   (
3     select driverid, forename, surname,
4            count(*) as races,
5            bool_and(position is null) as never_finished
6       from drivers
7            join results using(driverid)
8            join races using(raceid)
9   group by driverid
```

```
)
  select driverid, forename, surname, races
    from counts
   where never_finished
order by races desc;
```

Well, it turns out that we have a great number of cases in which it happens. The previous query gives us 202 drivers who never finished a single race they took part in, 117 of them had only participated in a single race that said.

Not picking on anyone in particular, we can find out if some seasons were less lucky than others on that basis and search for drivers who didn't finish a single race they participated into, per season:

```
with counts as
(
  select date_trunc('year', date) as year,
         count(*) filter(where position is null) as outs,
         bool_and(position is null) as never_finished
    from drivers
         join results using(driverid)
         join races using(raceid)
group by date_trunc('year', date), driverid
)
  select extract(year from year) as season,
         sum(outs) as "#times any driver didn't finish a race"
    from counts
   where never_finished
group by season
order by sum(outs) desc
   limit 5;
```

In this query, you can see the aggregate *filter(where . . .)* syntax that allows us to update our computation only for those rows that pass the filter. Here we choose to count all race results where the position is null, which means the driver didn't make it to the finish line for some reason. . .

```
 season | #times any driver didn't finish a race
════════╪════════════════════════════════════════
   1989 │                                    139
   1953 │                                     51
   1955 │                                     48
   1990 │                                     48
   1956 │                                     46
(5 rows)
```

It turns out that overall, 1989 was a pretty bad season.

Aggregates Without a Group By

It is possible to compute aggregates over a data set without using the *group by* clause in SQL. What it then means is that we are operating over a single group that contains the whole result set:

```
select count(*)
  from races;
```

This very simple query computes the count of all the races. It has built an implicit group of rows, containing everything.

Restrict Selected Groups: Having

Are you curious about the reasons why those drivers couldn't make it to the end of the race? I am too, so let's inquire about that!

```
\set season 'date ''1978-01-01'''

  select status, count(*)
    from results
         join races using(raceid)
         join status using(statusid)
   where date >= :season
     and date <  :season + interval '1 year'
     and position is null
group by status
  having count(*) >= 10
order by count(*) desc;
```

The query introduces the *having* clause. Its purpose is to filter the result set to only those groups that meet the *having* filtering condition, much as the *where* clause works for the individual rows selected for the result set.

Note that to avoid any ambiguity, the *having* clause is not allowed to reference *select* output aliases.

```
       status        | count
═════════════════════╪═══════
 Did not qualify     |    55
 Accident            |    46
 Engine              |    37
 Did not prequalify  |    25
 Gearbox             |    13
 Spun off            |    12
 Transmission        |    12
(7 rows)
```

We can see that drivers mostly do not finish a race because they didn't qualify to take part in it. Another quite common reason for not finishing is that the driver had an accident.

Grouping Sets

A restriction with classic aggregates is that you can only run them through a single group definition at a time. In some cases, you want to be able to compute aggregates for several groups in parallel. For those cases, SQL provides the *grouping sets* feature.

In the *Formula One* competition, points are given to drivers and then used to compute both the driver's champion and the constructor's champion points. Can we compute those two sums over the same points in a single query? Yes, of course, we can:

```
\set season 'date ''1978-01-01'''

    select drivers.surname as driver,
           constructors.name as constructor,
           sum(points) as points

      from results
           join races using(raceid)
           join drivers using(driverid)
           join constructors using(constructorid)

   where date >= :season
     and date <  :season + interval '1 year'

  group by grouping sets((drivers.surname),
                         (constructors.name))
    having sum(points) > 20

  order by constructors.name is not null,
           drivers.surname is not null,
           points desc;
```

And we get the following result:

```
  driver   │ constructor │ points
═══════════╪═════════════╪════════
 Andretti  │ ¤           │     64
 Peterson  │ ¤           │     51
 Reutemann │ ¤           │     48
 Lauda     │ ¤           │     44
 Depailler │ ¤           │     34
```

```
 Watson    | ¤           |     25
 Scheckter | ¤           |     24
 ¤         | Team Lotus  |    116
 ¤         | Brabham     |     69
 ¤         | Ferrari     |     65
 ¤         | Tyrrell     |     41
 ¤         | Wolf        |     24
(12 rows)
```

We see that we get *null* entries for drivers when the aggregate has been computed for a constructor's group and *null* entries for constructors when the aggregate has been computed for a driver's group.

Two other kinds of *grouping sets* are included in order to simplify writing queries. They are only syntactic sugarcoating on top of the previous capabilities.

The *rollup* clause generates permutations for each column of the *grouping sets*, one after the other. That's useful mainly for hierarchical data sets, and it is still useful in our Formula One world of champions. In the 80s Prost and Senna were all the rage, so let's dive into their results and points:

```
    select drivers.surname as driver,
           constructors.name as constructor,
           sum(points) as points

      from results
           join races using(raceid)
           join drivers using(driverid)
           join constructors using(constructorid)

     where drivers.surname in ('Prost', 'Senna')

  group by rollup(drivers.surname, constructors.name);
```

Given this query, in a single round-trip we fetch the cumulative points for Prost for each of the constructor's championship he raced for, so a total combined 798.5 points where the constructor is null. Then we do the same thing for Senna of course. And finally, the last line is the total amount of points for everybody involved in the result set.

```
 driver | constructor | points
════════╪═════════════╪════════
 Prost  | Ferrari     |    107
 Prost  | McLaren     |  458.5
 Prost  | Renault     |    134
 Prost  | Williams    |     99
 Prost  | ¤           |  798.5
```

```
 Senna  | HRT          |      0
 Senna  | McLaren      |    451
 Senna  | Renault      |      2
 Senna  | Team Lotus   |    150
 Senna  | Toleman      |     13
 Senna  | Williams     |     31
 Senna  | ¤            |    647
 ¤      | ¤            | 1445.5
(13 rows)
```

Another kind of *grouping sets* clause shortcut is named *cube*, which extends to all permutations available, including partial ones:

```
    select drivers.surname as driver,
           constructors.name as constructor,
           sum(points) as points

      from results
           join races using(raceid)
           join drivers using(driverid)
           join constructors using(constructorid)

     where drivers.surname in ('Prost', 'Senna')

  group by cube(drivers.surname, constructors.name);
```

Thanks to the cube here we can see both the total amount of points racked up by to those exceptional drivers over their entire careers. We have each driver's points by constructor, and when constructor is *NULL* we have the total amount of points for the driver. That's 798.5 points for Prost and 647 for Senna.

Also in the same query, we can see the points per constructor, independent of the driver, as both Prost and Senna raced for McLaren, Renault, and Williams at different times. And for two seasons, Prost and Senna both raced for McLaren, too.

```
 driver | constructor | points
════════╪═════════════╪════════
 Prost  | Ferrari     |    107
 Prost  | McLaren     |  458.5
 Prost  | Renault     |    134
 Prost  | Williams    |     99
 Prost  | ¤           |  798.5
 Senna  | HRT         |      0
 Senna  | McLaren     |    451
 Senna  | Renault     |      2
 Senna  | Team Lotus  |    150
 Senna  | Toleman     |     13
```

```
 Senna   | Williams     |     31
 Senna   | ¤            |    647
 ¤       | ¤            | 1445.5
 ¤       | Ferrari      |    107
 ¤       | HRT          |      0
 ¤       | McLaren      |  909.5
 ¤       | Renault      |    136
 ¤       | Team Lotus   |    150
 ¤       | Toleman      |     13
 ¤       | Williams     |    130
(20 rows)
```

Common Table Expressions: With

Earlier we saw many drivers who didn't finish the race because of accidents, and that was even the second reason listed just after *did not qualify*. This brings into question the level of danger in those Formula One races. How frequent is an accident in a Formula One competition? First we can have a look at the most dangerous seasons in terms of accidents.

```
  select extract(year from races.date) as season,
         count(*)
           filter(where status = 'Accident') as accidents

    from results
         join status using(statusid)
         join races using(raceid)

group by season
order by accidents desc
   limit 5;
```

So the five seasons with the most accidents in the history of Formula One are:

```
 season | accidents
════════╪═══════════
   1977 |        60
   1975 |        54
   1978 |        48
   1976 |        48
   1985 |        36
(5 rows)
```

It seems the most dangerous seasons of all time are clustered at the end of the 70s and the beginning of the 80s, so we are going to zoom in on this period with the following console friendly histogram query:

```
with accidents as
 (
    select extract(year from races.date) as season,
           count(*) as participants,
           count(*) filter(where status = 'Accident') as accidents
      from results
           join status using(statusid)
           join races using(raceid)
  group by season
 )
  select season,
         round(100.0 * accidents / participants, 2) as pct,
         repeat(text '■',
                ceil(100*accidents/participants)::int
               )
         as bar
    from accidents
   where season between 1974 and 1990
order by season;
```

Common table expression is the full name of the *with* clause that you see in effect in the query. It allows us to run a subquery as a prologue, and then refer to its result set like any other relation in the *from* clause of the main query. In our case, you can see that the main query is doing *from accidents*, and the *CTE* has been given that name.

In the *accidents* CTE we compute basic information such as how many participants we had overall in all the races of each season (we know this is the number of lines in the result table for the races that happened in the selected year, so that's the `count(*)` column — and we also compute how many of those participants had an accident, thanks to the *filter* clause that we introduced before.

Given the *accident* relation from the *CTE*, it is then easy to compute a percentage of accidents over race participants, and we can even get fancy and display the percentage in the form of a horizontal bar diagram by repeating a unicode black square character so that we have a fancy display:

season	pct	bar
1974	3.67	■■■
1975	14.88	■■■■■■■■■■■■■■■
1976	11.06	■■■■■■■■■■■
1977	12.58	■■■■■■■■■■■■
1978	10.19	■■■■■■■■■■
1979	7.20	■■■■■■■
1980	7.83	■■■■■■■

```
   1981 |  3.56 | ■■■
   1982 |  0.86 |
   1983 |  0.00 |
   1984 |  5.58 | ■■■■■
   1985 |  8.87 | ■■■■■■■■
   1986 |  6.07 | ■■■■■■
   1987 |  5.97 | ■■■■■
   1988 |  0.61 |
   1989 |  0.81 |
   1990 |  1.29 | ■
(17 rows)
```

The Formula One racing seems to be interesting enough outside of what we cover in this book and the respective database: Wikipedia is full of information about this sport. In the list of Formula One seasons, we can see a table of all seasons and their champion driver and champion constructor: the driver/constructor who won the most points in total in the races that year.

To compute that in SQL we need to first add up the points for each driver and constructor and then we can select those who won the most each season:

```
with points as
 (
    select year as season, driverid, constructorid,
           sum(points) as points
      from results join races using(raceid)
  group by grouping sets((year, driverid),
                         (year, constructorid))
    having sum(points) > 0
  order by season, points desc
 ),
 tops as
 (
    select season,
           max(points) filter(where driverid is null) as ctops,
           max(points) filter(where constructorid is null) as dtops
      from points
  group by season
  order by season, dtops, ctops
 ),
 champs as
 (
    select tops.season,
           champ_driver.driverid,
           champ_driver.points,
           champ_constructor.constructorid,
           champ_constructor.points
```

```
27
28        from tops
29             join points as champ_driver
30               on champ_driver.season = tops.season
31              and champ_driver.constructorid is null
32              and champ_driver.points = tops.dtops
33
34             join points as champ_constructor
35               on champ_constructor.season = tops.season
36              and champ_constructor.driverid is null
37              and champ_constructor.points = tops.ctops
38    )
39    select season,
40           format('%s %s', drivers.forename, drivers.surname)
41              as "Driver's Champion",
42           constructors.name
43              as "Constructor's champion"
44      from champs
45           join drivers using(driverid)
46           join constructors using(constructorid)
47  order by season;
```

This time we get about a full page SQL query, and yes it's getting complex. The main thing to see is that we are *daisy chaining* the CTEs:

1. The *points* CTE is computing the sum of points for both the drivers and the constructors for each season.

   We can do that in a single SQL query thanks to the *grouping sets* feature that is covered in more details later in this book. It allows us to run aggregates over more than one group at a time within a single query scan.

2. The *tops* CTE is using the *points* one in its *from* clause and it computes the maximum points any driver and constructor had in any given season,

   We do that in a separate step because in SQL it's not possible to compute an aggregate over an aggregate:

   ```
   ERROR:  aggregate function calls cannot be nested
   ```

   Thus the way to have the sum of points and the maximum value for the sum of points in the same query is by using a two-stages pipeline, which is what we are doing.

3. The *champs* CTE uses the *tops* and the *points* data to restrict our result set to the champions, that is those drivers and constructors

who made as many points as the maximum.

Additionnaly, in the *champs* CTE we can see that we use the *points* data twice for different purposes, aliasing the relation to *champ_driver* when looking for the champion driver, and to *champ_constructor* when looking for the champion constructor.

4. Finally we have the outer query that uses the *champs* dataset and formats it for the application, which is close to what our Wikipedia example page is showing.

Here's a cut-down version of the 68 rows in the final result set:

```
 season │ Driver's Champion  │ Constructor's champion
════════╪════════════════════╪════════════════════════
   1950 │ Nino Farina        │ Alfa Romeo
   1951 │ Juan Fangio        │ Ferrari
   1952 │ Alberto Ascari     │ Ferrari
   1953 │ Alberto Ascari     │ Ferrari
   1954 │ Juan Fangio        │ Ferrari
   1955 │ Juan Fangio        │ Mercedes
   1956 │ Juan Fangio        │ Ferrari
   1957 │ Juan Fangio        │ Maserati
...
   1985 │ Alain Prost        │ McLaren
   1986 │ Alain Prost        │ Williams
   1987 │ Nelson Piquet      │ Williams
   1988 │ Alain Prost        │ McLaren
   1989 │ Alain Prost        │ McLaren
   1990 │ Ayrton Senna       │ McLaren
   1991 │ Ayrton Senna       │ McLaren
   1992 │ Nigel Mansell      │ Williams
   1993 │ Alain Prost        │ Williams
...
   2013 │ Sebastian Vettel   │ Red Bull
   2014 │ Lewis Hamilton     │ Mercedes
   2015 │ Lewis Hamilton     │ Mercedes
   2016 │ Nico Rosberg       │ Mercedes
```

Distinct On

Another useful PostgreSQL extension is the *distinct on* SQL form, and here's what the PostgreSQL distinct clause documentation has to say about it:

> SELECT DISTINCT ON (expression [, ...]) keeps only the first row of each set of rows where the given expressions evalu-

> ate to equal. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY (see above). Note that the "first row" of each set is unpredictable unless ORDER BY is used to ensure that the desired row appears first.

So it is possible to return the list of drivers who ever won a race in the whole Formula One history with the following query:

```
select distinct on (driverid)
       forename, surname
  from results
       join drivers using(driverid)
 where position = 1;
```

There 107 of them, as we can check with the following query:

```
select count(distinct(driverid))
  from results
       join drivers using(driverid)
 where position = 1;
```

The classic way to have a single result per driver in SQL would be to aggregate over them, creating a group per driver:

```
  select forename, surname
    from results join drivers using(driverid)
   where position = 1
group by drivers.driverid;
```

Note that we are using the *group by* clause without aggregates. That's a valid use case for this clause, allowing us to force unique entries per group in the result set.

Result Sets Operations

SQL also includes set operations for combining queries results sets into a single one.

In our data model we have a *driverstandings* and a *constructorstandings* — they contain data that come from the *results* table that we've been using a lot, so that you can query a smaller data set... or I guess so that you can write simple SQL queries.

The set operations are *union*, *intersect* and *except*. As expected with *union* you can assemble a result set from the result of several queries:

```
(
      select raceid,
             'driver' as type,
             format('%s %s',
                    drivers.forename,
                    drivers.surname)
             as name,
             driverstandings.points

        from driverstandings
             join drivers using(driverid)

       where raceid = 972
         and points > 0
)
union all
(
      select raceid,
             'constructor' as type,
             constructors.name as name,
             constructorstandings.points

        from constructorstandings
             join constructors using(constructorid)

       where raceid = 972
         and points > 0
)
order by points desc;
```

Here, in a single query, we get the list of points from race 972 for drivers and constructors, well anyway all of them who got points. It is a classic of using *union*, as we are adding static column values in each branch of the query, so that we know where each line of the result set comes from:

raceid	type	name	points
972	constructor	Mercedes	136
972	constructor	Ferrari	135
972	driver	Sebastian Vettel	86
972	driver	Lewis Hamilton	73
972	driver	Valtteri Bottas	63
972	constructor	Red Bull	57
972	driver	Kimi Räikkönen	49
972	driver	Max Verstappen	35
972	constructor	Force India	31
972	driver	Daniel Ricciardo	22
972	driver	Sergio Pérez	22
972	constructor	Williams	18
972	driver	Felipe Massa	18

```
    972 | constructor | Toro Rosso       |     13
    972 | driver      | Carlos Sainz     |     11
    972 | driver      | Esteban Ocon     |      9
    972 | constructor | Haas F1 Team     |      8
    972 | driver      | Nico Hülkenberg  |      6
    972 | constructor | Renault          |      6
    972 | driver      | Romain Grosjean  |      4
    972 | driver      | Kevin Magnussen  |      4
    972 | driver      | Daniil Kvyat     |      2
(22 rows)
```

In our writing of the query, you may notice that we did parenthesize the branches of the *union*. It's not required that we do so, but it improves the readability of the query and makes it obvious as to what data set the *order by* clause is applied for.

Finally, we've been using *union all* in this query. That's because the way the queries are built is known to never yield duplicates in the result set. It may happen that you need to use a *union* query and then want to remove duplicates from the result set, that's what *union* (with no *all*) does.

The next query is a little convoluted and lists the drivers who received no points in race 972 (Russian Grand Prix of 2017-04-30) despite having gotten some points in the previous race (id 971, Bahrain Grand Prix of 2017-04-16):

```
(
      select driverid,
             format('%s %s',
                    drivers.forename,
                    drivers.surname)
             as name

        from results
             join drivers using(driverid)

       where raceid = 972
         and points = 0
)
except
(
      select driverid,
             format('%s %s',
                    drivers.forename,
                    drivers.surname)
             as name

        from results
             join drivers using(driverid)
```

```
24 |
25 |       where raceid = 971
26 |         and points = 0
27 | )
28 | ;
```

Which gives us:

```
 driverid |       name
══════════╪══════════════════
      154 | Romain Grosjean
      817 | Daniel Ricciardo
(2 rows)
```

Here it's also possible to work with the *intersect* operator in between result sets. With our previous query, we would get the list of drivers who had no points in either race.

The *except* operator is very useful for writing test cases, as it allows us to compute a difference in between two result sets. One way to use it is to store the result of running a query against a known *fixture* or database content in an expected file. Then when you change a query, it's easy to load your expected data set into the database and compare it with the result of running the new query.

We said earlier that the following two queries are supposed to return the same dataset, so let's check that out:

```
1  | (
2  |       select name, location, country
3  |         from circuits
4  |     order by position <-> point(2.349014, 48.864716)
5  | )
6  | except
7  | (
8  |       select name, location, country
9  |         from circuits
10 |     order by point(lng,lat) <-> point(2.349014, 48.864716)
11 | )
12 | ;
```

This returns 0 rows, so the index is reliable *and* the *location* column is filled with the same data as found in the *lng* and *lat* columns.

You can implement some regression testing pretty easily thanks to the *except* operator!

Understanding Nulls

Given its relational theory background, SQL comes with a special value that has no counterpart in a common programming language: *null*. In Python, we have *None*, in PHP we have *null*, in C we have *nil*, and about every other programming language has something that looks like a *null*.

Three-Valued Logic

The difference in SQL is that *null* introduces ***three-valued logic***. Where that's very different from other languages *None* or *Null* is when comparing values. Let's have a look at the SQL *null* truth table:

```
select a::text, b::text,
       (a=b)::text as "a=b",
       format('%s = %s',
              coalesce(a::text, 'null'),
              coalesce(b::text, 'null')) as op,
       format('is %s',
              coalesce((a=b)::text, 'null')) as result
  from (values(true), (false), (null)) v1(a)
       cross join
       (values(true), (false), (null)) v2(b);
```

As you can see *cross join* is very useful for producing a truth table. It implements a Cartesian product over our columns, here listing the first value of *a* (*true*) with every value of *b* in order (*true*, then *false*, then *null*), then again with the second value of *a* (*false*) and then again with the third value of *a* (*null*).

We are using *format* and *coalesce* to produce an easier to read results table here. The *coalesce* function returns the first of its argument which is not null, with the restriction that all of its arguments must be of the same data type, here *text*. Here's the nice truth table we get:

a	b	a=b	op	result
true	true	true	true = true	is true
true	false	false	true = false	is false
true	¤	¤	true = null	is null
false	true	false	false = true	is false
false	false	true	false = false	is true
false	¤	¤	false = null	is null

```
 ¤    | true  | ¤    | null = true   | is null
 ¤    | false | ¤    | null = false  | is null
 ¤    | ¤     | ¤    | null = null   | is null
(9 rows)
```

We can think of *null* as meaning *I don't know what this is* rather than *no value here*. Say you have in A (left hand) something (hidden) that you don't know what it is and in B (right hand) something (hidden) that you don't know what it is. You're asked if A and B are the same thing. Well, you can't know that, can you?

So in SQL *null = null* returns *null*, which is the proper answer to the question, but not always the one you expect, or the one that allows you to write your query and have the expected result set.

That's why we have other SQL operators to work with data that might be *null*: they are *is distinct from* and *is not distinct from*. Those two operators not only have a very long name, they also pretend that *null* is the same thing as *null*.

So if you want to pretend that SQL doesn't implement three-valued logic you can use those operators and forget about Boolean comparisons returning *null*.

We can even easily obtain the *truth table* from a SQL query directly:

```
select a::text as left, b::text as right,
       (a = b)::text as "=",
       (a <> b)::text as "<>",
       (a is distinct from b)::text as "is distinct",
       (a is not distinct from b)::text as "is not distinct from"
  from            (values(true),(false),(null)) t1(a)
       cross join (values(true),(false),(null)) t2(b);
```

With this complete result this time:

```
 left  | right |   =   |  <>   | is distinct | is not distinct from
═══════╪═══════╪═══════╪═══════╪═════════════╪══════════════════════
 true  | true  | true  | false | false       | true
 true  | false | false | true  | true        | false
 true  | ¤     | ¤     | ¤     | true        | false
 false | true  | false | true  | true        | false
 false | false | true  | false | false       | true
 false | ¤     | ¤     | ¤     | true        | false
 ¤     | true  | ¤     | ¤     | true        | false
 ¤     | false | ¤     | ¤     | true        | false
 ¤     | ¤     | ¤     | ¤     | false       | true
(9 rows)
```

You can see that we have not a single *null* in the last two columns.

Not Null Constraints

In some cases, in your data model you want the strong guarantee that a column cannot be *null*. Usually that's because it makes no sense for your application to deal with some *unknowns*, or in other words, you are dealing with a **required** value.

The default value for any column, unless you specify something else, is always *null*. It's only a default value though, it's not a constraint on your data model, so your application may insert a *null* value in a column with a *non null* default:

```
create table test(id serial, f1 text default 'unknown');
insert into test(f1) values(DEFAULT),(NULL),('foo');
table test;
```

This script gives the following output:

```
 id |   f1
════╪═══════
  1 | unknown
  2 | ¤
  3 | foo
```

As we can see, we have a *null* value in our test table, despite having implemented a specific default value. The way to avoid that is using a *not null* constraint:

```
drop table test;
create table test(id serial, f1 text not null default 'unknown');
insert into test(f1) values(DEFAULT),(NULL),('foo');
ERROR:  null value in column "f1" violates not-null constraint
DETAIL:  Failing row contains (2, null).
```

This time the *insert* command fails: accepting the data would violate the constraint we specified at table creation, i.e. no *null* allowed.

Outer Joins Introducing Nulls

As we saw earlier in this chapter, outer joins are meant to preserve rows from your reference relation and add to it columns from the outer relation

when the join condition is satisfied. When the join condition is not satisfied, the outer joins then fill the columns from the outer relation with *null* values.

A typical example would be with calendar dates when we have not registered data at given dates yet. In our motor racing database example, we can ask for the name of the pole position's driver and the final position. As the model registers the races early, some of them won't have run yet and so the results are not available in the database:

```
  select races.date,
         races.name,
         drivers.surname as pole_position,
         results.position
    from races
         /*
          * We want only the pole position from the races
          * know the result of and still list the race when
          * we don't know the results.
          */
         left join results
                on races.raceid = results.raceid
               and results.grid = 1
         left join drivers using(driverid)
   where      date >= '2017-05-01'
         and date < '2017-08-01'
order by races.date;
```

So we can see that we only have data from races before the 25 June in the version that was used to prepare this book:

```
    date    |         name          | pole_position | position
════════════╪═══════════════════════╪═══════════════╪══════════
 2017-05-14 | Spanish Grand Prix    | Hamilton      |        1
 2017-05-28 | Monaco Grand Prix     | Räikkönen     |        2
 2017-06-11 | Canadian Grand Prix   | Hamilton      |        1
 2017-06-25 | Azerbaijan Grand Prix | ¤             |        ¤
 2017-07-09 | Austrian Grand Prix   | ¤             |        ¤
 2017-07-16 | British Grand Prix    | ¤             |        ¤
 2017-07-30 | Hungarian Grand Prix  | ¤             |        ¤
(7 rows)
```

With *grid* having a *not null* constraints in your database model for the *results* table, we see that sometimes we don't have the data at all. Another way to say that we don't have the data is to say that we don't know the answer to the query. In this case, SQL uses *null* in its answer.

So *null* values can be created by the queries themselves. There's basically

no way to escape from having to deal with *null* values, so your application must be prepared for them and moreover understand what to do with them.

Using Null in Applications

Most programming languages come with a representation of the unknown or not yet initialized state, be it *None* in Python, *null* in Java and *C* and *PHP* and others, with varying semantics, or even the Ocaml option type or the Haskell maybe type.

Depending on your tools of choice the *null* SQL value maps quite directly to those concepts. The main thing is then to remember that you might get *null* in your results set, and you should write your code accordingly. The next main thing to keep in mind is the three-valued logic semantics when you write SQL, and remember to use `where foo is null` if that's what you mean, rather than the erroneous `where foo = null`, because `null = null is null` and then it won't be selected in your resultset:

```
select a, b
  from (values(true), (false), (null)) v1(a)
       cross join
       (values(true), (false), (null)) v2(b)
where a = null;
```

That gives nothing, as we saw before, as there's no such row where anything equals null:

```
 a | b
═══╪═══
(0 rows)
```

Now if you remember your logic, then you can instead ask the right question:

```
select a, b
  from (values(true), (false), (null)) v1(a)
       cross join
       (values(true), (false), (null)) v2(b)
where a is null;
```

You then obtain those rows for which *a is null*:

```
 a | b
═══╪═══
 ¤ | t
```

```
 ¤ | f
 ¤ | ¤
(3 rows)
```

Understanding Window Functions

There was SQL before window functions and there is SQL after *window functions*: that's how powerful this tool is!

The whole idea behind *window functions* is to allow you to process several values of the result set at a time: you see through the window some *peer* rows and you are able to compute a single output value from them, much like when using an *aggregate* function.

Windows and Frames

PostgreSQL comes with plenty of features, and one of them will be of great help when it comes to getting a better grasp of what's happening with *window functions*. The first step we are going through here is understanding what data the function has access to. For each input row, you have access to a frame of the data, and the first thing to understand here is that *frame*.

The `array_agg()` function is an *aggregate* function that builds an array. Let's use this tool to understand *window frames*:

```
1 | select x, array_agg(x) over (order by x)
2 |   from generate_series(1, 3) as t(x);
```

The *array_agg()* aggregates every value in the current frame, and here outputs the full exact content of the *windowing* we're going to process.

```
 x | array_agg
---+-----------
 1 | {1}
 2 | {1,2}
 3 | {1,2,3}
(3 rows)
```

The window definition `over (order by x)` actually means `over (order by x rows between unbounded preceding and current row)`:

```
1 | select x,
2 |        array_agg(x) over (order by x
```

```
                              rows between unbounded preceding
                                       and current row)
   from generate_series(1, 3) as t(x);
```

And of course we get the same result set as before:

```
 x | array_agg
---+-----------
 1 | {1}
 2 | {1,2}
 3 | {1,2,3}
(3 rows)
```

It's possible to work with other kinds of *frame specifications* too, as in the following examples:

```
select x,
       array_agg(x) over (rows between current row
                                   and unbounded following)
  from generate_series(1, 3) as t(x);
```

```
 x | array_agg
---+-----------
 1 | {1,2,3}
 2 | {2,3}
 3 | {3}
(3 rows)
```

If no frame clause is used at all, then the default is too see the whole set of rows in each of them, which can be really useful if you want to compute sums and percentages for example:

```
select x,
       array_agg(x) over () as frame,
       sum(x) over () as sum,
       x::float/sum(x) over () as part
  from generate_series(1, 3) as t(x);
```

```
 x |  frame  | sum |       part
---+---------+-----+-------------------
 1 | {1,2,3} |   6 | 0.166666666666667
 2 | {1,2,3} |   6 | 0.333333333333333
 3 | {1,2,3} |   6 |               0.5
(3 rows)
```

Did you know you could compute both the total sum of a column and the ratio of the current value compared to the this total within a single SQL query? That's the breakthrough we're talking about now with *window functions*.

Partitioning into Different Frames

Other frames are possible to define when using the clause `PARTITION BY`. It allows defining as *peer rows* those rows that share a common property with the *current row*, and the property is defined as a *partition*.

So in the *Formula One* database we have a *results* table with results from all the known races. Let's pick a race:

```
-[ RECORD 1 ]----------------------------------------------------
raceid    | 890
year      | 2013
round     | 10
circuitid | 11
name      | Hungarian Grand Prix
date      | 2013-07-28
time      | 12:00:00
url       | http://en.wikipedia.org/wiki/2013_Hungarian_Grand_Prix
```

Within that race, we can now fetch the list of competing drivers in their position order (winner first), and also their ranking compared to other drivers from the same constructor in the race:

```
select surname,
         constructors.name,
         position,
         format('%s / %s',
                row_number()
                  over(partition by constructorid
                           order by position nulls last),

                count(*) over(partition by constructorid)
               )
            as "pos same constr"
    from      results
         join drivers using(driverid)
         join constructors using(constructorid)
   where raceid = 890
order by position;
```

The *partition by* frame allows us to see *peer rows*, here the rows from *results* where the *constructorid* is the same as the current row. We use that partition twice in the previous SQL query, in the `format()` call. The first time with the `row_number()` window function gives us the position in the race with respect to other drivers from the same constructor, and the second time with `count(*)` gives us how many drivers from the same constructor participated in the race:

```
    surname    |     name      | position | pos same constr
```

Hamilton	Mercedes	1	1 / 2
Räikkönen	Lotus F1	2	1 / 2
Vettel	Red Bull	3	1 / 2
Webber	Red Bull	4	2 / 2
Alonso	Ferrari	5	1 / 2
Grosjean	Lotus F1	6	2 / 2
Button	McLaren	7	1 / 2
Massa	Ferrari	8	2 / 2
Pérez	McLaren	9	2 / 2
Maldonado	Williams	10	1 / 2
Hülkenberg	Sauber	11	1 / 2
Vergne	Toro Rosso	12	1 / 2
Ricciardo	Toro Rosso	13	2 / 2
van der Garde	Caterham	14	1 / 2
Pic	Caterham	15	2 / 2
Bianchi	Marussia	16	1 / 2
Chilton	Marussia	17	2 / 2
di Resta	Force India	18	1 / 2
Rosberg	Mercedes	19	2 / 2
Bottas	Williams	¤	2 / 2
Sutil	Force India	¤	2 / 2
Gutiérrez	Sauber	¤	2 / 2

(22 rows)

In a single SQL query, we can obtain information about each driver in the race and add to that other information from the race as a whole. Remember that the *window functions* only happens after the *where* clause, so you only get to see rows from the available result set of the query.

Available Window Functions

Any and all *aggregate* function you already know can be used against a *window frame* rather than a *grouping clause*, so you can already start to use *sum*, *min*, *max*, *count*, *avg*, and the other that you're already used to using.

You might already know that with PostgreSQL it's possible to use the CREATE AGGREGATE command to register your own *custom aggregate*. Any such custom aggregate can also be given a *window frame definition* to work on.

PostgreSQL of course is included with built-in aggregate functions and a number of built-in window functions.

```
  select surname,
         position as pos,
         row_number()
           over(order by fastestlapspeed::numeric)
           as fast,
         ntile(3) over w as "group",
         lag(code, 1) over w as "prev",
         lead(code, 1) over w as "next"
    from      results
         join drivers using(driverid)
   where raceid = 890
  window w as (order by position)
order by position;
```

In this example you can see that we are reusing the same *window definition* several times, so we're giving it a name to simplify the SQL. In this query we are fetching for each driver we are fetching his position in the results, his position in terms of *fastest lap speed*, a *group* number if we divide the drivers into a set of four groups thanks to the *ntile* function, the name of the previous driver who made it, and the name of the driver immediately next to the current one, thanks to the *lag* an *lead* functions:

```
    surname     | pos | fast | group | prev | next
════════════════╪═════╪══════╪═══════╪══════╪══════
 Hamilton       |   1 |   20 |     1 | ¤    | RAI
 Räikkönen      |   2 |   17 |     1 | HAM  | VET
 Vettel         |   3 |   21 |     1 | RAI  | WEB
 Webber         |   4 |   22 |     1 | VET  | ALO
 Alonso         |   5 |   15 |     1 | WEB  | GRO
 Grosjean       |   6 |   16 |     1 | ALO  | BUT
 Button         |   7 |   12 |     1 | GRO  | MAS
 Massa          |   8 |   18 |     1 | BUT  | PER
 Pérez          |   9 |   13 |     2 | MAS  | MAL
 Maldonado      |  10 |   14 |     2 | PER  | HUL
 Hülkenberg     |  11 |    9 |     2 | MAL  | VER
 Vergne         |  12 |   11 |     2 | HUL  | RIC
 Ricciardo      |  13 |    8 |     2 | VER  | VDG
 van der Garde  |  14 |    6 |     2 | RIC  | PIC
 Pic            |  15 |    5 |     2 | VDG  | BIA
 Bianchi        |  16 |    3 |     3 | PIC  | CHI
 Chilton        |  17 |    4 |     3 | BIA  | DIR
 di Resta       |  18 |   10 |     3 | CHI  | ROS
 Rosberg        |  19 |   19 |     3 | DIR  | BOT
 Sutil          |   ¤ |    2 |     3 | GUT  | ¤
 Gutiérrez      |   ¤ |    1 |     3 | BOT  | SUT
 Bottas         |   ¤ |    7 |     3 | ROS  | GUT
(22 rows)
```

And we can see that the *fastest lap speed* is not as important as one might

think, as both the two fastest drivers didn't even finish the race. In SQL terms we also see that we can have two different sequences returned from the same query, and again we can reference other rows.

When to Use Window Functions

The real magic of what are called *window functions* is actually the frame of data they can see when using the `OVER ()` clause. This frame is specified thanks to the `PARTITION BY` and `ORDER BY` clauses.

You need to remember that the windowing clauses are always considered last in the query, meaning after the *where* clause. In any frame You can only see rows that have been selected for output: e.g. it's not directly possible to compute a percentage of values that you don't want to display. You would need to use a subquery in that case.

Use *window functions* whenever you want to compute values for each row of the result set and those computations depend on other rows within the same result set. A classic example is a marketing analysis of weekly results: you typically output both each day's gross sales and the variation with the same day in comparison to the previous week.

Understanding Relations and Joins

In the previous section, we saw some bits about data sources in SQL when introducing the *from* clause and some join operations. In this section we are going to expand this on this part and look specifically at what a relation is.

As usual, the PostgreSQL documentation provides us with some enlightenment us (here in its section entitled The FROM Clause:

> A table reference can be a table name (possibly schema-qualified), or a derived table such as a subquery, a JOIN construct, or complex combinations of these. If more than one table reference is listed in the FROM clause, the tables are cross-joined (that is, the Cartesian product of their rows is formed; see below). The result of the FROM list

is an intermediate virtual table that can then be subject to transformations by the WHERE, GROUP BY, and HAVING clauses and is finally the result of the overall table expression.

Relations

We already know that a relation is a set of data all having a common set of properties, that to say a set of elements all from the same composite data type. The SQL standard didn't go as far as defining relations in terms of being a set in the mathematical way of looking at it, and that would imply that no duplicates are allowed. We can then talk about a bag rather than a set, because duplicates are allowed in SQL relations.

The data types are defined either by the *create type* statement or by the more common *create table* statement:

```
~# create table relation(id integer, f1 text, f2 date, f3 point);
CREATE TABLE

~# insert into relation
        values(1,
               'one',
               current_date,
               point(2.349014, 48.864716)
              );
INSERT 0 1

~# select relation from relation;
                 relation
═══════════════════════════════════════════
 (1,one,2017-07-04,"(2.349014,48.864716)")
(1 row)
```

Here we created a table named *relation*. What happens in the background is that PostgreSQL created a type with the same name that you can manipulate, or reference. So the *select* statement here is returning tuples of the composite type *relation*.

SQL is a strongly typed programming language: at query planning time the data type of every column of the result set must be known. Any result set is defined in terms of being a *relation* of a known composite data type, where each and every row in the result set shares the common properties implied by this data type.

The relations can be defined in advance in *create table* or *create type* statements, or defined on the fly by the query planner when it makes sense for your query. Other statements can also create data types too, such as *create view* — more on that later.

When you use a subquery in your main query, either in the form of a *common table expression* or directly inlined in your *from* clause, you are effectively defining a relation data type. At query run time, this relation is filled with a dataset, thus you have a full-blown relation to use.

Relational algebra is thereby a formalism of what you can do with such things. In short, this means joins. The result of a join in between two relations is a relation, of course, and that relation can in-turn participates into other *join* operations.

The result of a *from* clause is a relation, withh which the query planner is executing the rest of your query: the *where* clause to restrict the relation dataset to what's interesting for the query, and other clauses, up until the *window functions* and the *select* projection are computed so that we can finally construct the result set, i.e. a relation.

The PostgreSQL optimizer will then re-arrange the computations needed so they're as efficient as possible, rather than doing things in the way they are written. This is much like when *gcc* is doing its magic and you can't even recognize your intentions when reading the assembly outcome, except that with PostgreSQL you can actually make sense of the *explain plan* for your query, and relate it to the query text you wrote.

SQL Join Types

Joins are the basic operations you do with relations. The nature of a join is to build a new relation from a pair of existing ones. The most basic join is a *cross join* or Cartesian product, as we saw in the Boolean truth table, where we built a result set of all possible combinations of all entries.

Other kinds of join associate data between the two relations that participate in the operation. The association is specified precisely in the *join condition* and is usually based on some equality operator, but it is not limited to that.

We might want to count how many drivers made it to the finish behind the

current one in any single race, as that's a good illustration of a non-equality join condition:

```
   select results.positionorder as position,
          drivers.code,
          count(behind.*) as behind

    from results
              join drivers using(driverid)

         left join results behind
                on results.raceid = behind.raceid
               and results.positionorder < behind.positionorder

   where results.raceid = 972
     and results.positionorder <= 3

group by results.positionorder, drivers.code
order by results.positionorder;
```

Here are our top three, with how many drivers found behind. We are using the *positionorder* column here because it attributes a position to drivers who didn't finish the race, which is useful for us in this very query:

```
 position | code | behind
══════════╪══════╪════════
        1 | BOT  |     19
        2 | VET  |     18
        3 | RAI  |     17
(3 rows)
```

In this example query, we can also see that we are using the same relation twice in the same *FROM* query, thus giving the relation different aliases. It would be tempting to name those aliases *r1* and *r2*, but much as you would not do that in your code when naming variables, it's best to give meaningful names to your the SQL objects in your queries.

Relational algebra includes set-based operations, and what we have in SQL are inner and outer joins, cross joins and lateral joins. We saw all of them in this chapter's example queries, and here's a quick summary:

- *Inner joins* are useful when you want to only keep rows that satisfy the *join condition* for both involved relation.
- *Outer joins* are useful when you want to keep a reference relation's dataset no matter what and enrich it with the dataset from the other relation when the *join condition* is satisfied.

The relation of which you want to keep all the rows of is pointed to in the name of the outer join, so it's written on the left-hand side in a *left join* and on the right-hand side in a *right join*.

When the *join condition* is not satisfied, it means you keep some known data and must fill in the result relation with data that doesn't exist, so that's when *null* is very useful, and also why *null* is a member of every SQL data type (including the Boolean data type),

- *Full outer joins* is a special case of an outer join where you want to keep all the rows in the dataset, whether they satisfy the join condition or not.
- *Lateral joins* introduce the capability for the *join condition* to be *pushed down* into the relation on the right, allowing for new semantics such as top-N queries, thanks to being able to use *limit* in a lateral subquery.

The key here is to remember that a join takes two relations and a join condition as input and it returns another relation. A relation here is a bag of rows that all share a common relation data type definition, known at query planning time.

An Interview with Markus Winand

Markus Winand is the author of the very famous book "SQL Performance explained" and he also provides both http://use-the-index-luke.com and http://modern-sql.com. Markus is a master of the SQL standard and he is a wizard in terms of how to use SQL to enable fast application delivery and solid run-time performances!

Figure 4.2: Use The Index, Luke!

Developers often say that SQL is hard to master. Do you agree? What would be your recommendations for them to improve their SQL skills?

I think the reason many people find SQL hard to learn is that it is a declarative programming language.

Most people first learn imperative programming: they put a number of instructions into a particular order so that their execution delivers the desired result. An SQL statement is different because it simply defines the result. This becomes most obvious in the select clause, which literally defines the columns of the result. Most of the other main clauses describe which rows should be present in the result. It is important to understand that the author of an SQL statement does not instruct the database how to run the query. That's up to the database to figure out.

So I think the most important step in mastering SQL is to stop thinking in imperative terms. One recurring example I've seen in the field is how people imagine that joins work and more specifically, which indexes can help in improving join performance. People constantly try to apply their knowledge about algorithms to SQL statements, without knowing which algorithm the database actually uses. This causes a lot of problems, confusion and frustration.

First, always focus on writing a clear statement to describe each column and row of the desired result. If needed, you can take care of performance afterwards. This however, requires some understanding of database internals.

What would you say is the ideal SQL wizardry level a developer should reach to be able to do their job correctly?

Knowing everything would be best, I guess ;)

In reality, hardly any programmer is just an SQL programmer. Most are Java, C#, PHP, or whatever programmers who — more or less frequently — use SQL to interact with a database. Obviously, not all of them need to be SQL experts.

Today's programming often boils down to choosing the right

tool for each problem. To do this job correctly, as you properly phrased it, programmers should at least know what their SQL database could do. Once you remember that SQL can do aggregations without group by—e.g. for running totals, moving averages, etc.—it's easy to search the Internet for the syntax. So I'd say every programmer (and even more so architects) should have a good overview of what SQL can do nowadays in order to recognize situations in which SQL offers the best solution.

Quite often, a few lines of SQL can replace dozens of lines of an imperative program. Most of the time, the SQL solution is more correct and even faster. In the vain of an old saying about shell scripts, I'd say: "Watch out or I'll replace a day's worth of your imperative programming with a very small SQL statement".

You know the detailed behavior of many different RDBMS engines and you are used to working with them. Would you write portable SQL code in applications or pick one engine and then use it to its full capacity, writing tailored SQL (both schema and queries)?

I first aim to use standard SQL. This is just because I know standard SQL best and I believe that the semantics of standard SQL have the most rigid definitions. That means standard SQL defines a meaningful behavior, even for the most obscure corner cases. Vendor extensions have a tendency to focus on the main cases. For corner cases, they might behave in surprising and inconsistent ways — just because nobody thought about that during specification.

Sometimes, I cannot solve a problem with standard SQL — at least not in a sufficiently elegant and efficient way. That is more often because the database at hand doesn't support the standard features that I'd like to use for this problem. However, sometimes the standard just doesn't provide the required functionality. In either case I'm also happy to use a vendor extension. For me, this is really just my personal order of preference for solving a problem — it is not a limitation in any way.

When it comes to the benefits of writing portable SQL, there seems to be a common misconception in the field. Quite often, people argue that they don't need portability because they will never use another database. And I actually agree with that argument in the sense that aiming for full portability does not make any sense if you don't need to run the software on different database right now.

On the other hand, I believe that portability is not only about the code — it is also about the people. I'd say it is even more about the people. If you use standard SQL by default and only revert to proprietary syntax if needed, the SQL statements will be easier for other people to understand, especially people used to another database. On the scale of the whole industry it means that bringing new personnel on board involves less friction. Even from the personal viewpoint of a single developer, it has a big benefit: if you are used to writing standard SQL then the chances increase that you can write SQL that works on many databases. This makes you more valuable in the job market.

However, there is one big exception and that's DDL – i.e. create statements. For DDL, I don't even aim for portability in the first place. This is pointless and too restricting. If you need to create tables, views, indexes, and the like for different databases, it is better to just maintain a separate schema definition for each of them.

How do you see PostgreSQL in the RDBMS offering?

PostgreSQL is in a very strong position. I keep on saying that from a developer's perspective, PostgreSQL's feature set is closer to that of a commercial database than to that of the open-source competitors such as MySQL/MariaDB.

I particularly like the rich standard SQL support PostgreSQL has: that means simple things like the fully featured values clause, but also `with [recursive]`, `over`, `lateral` *and arrays.*

5 Data Types

Contents

Reading the Wikipedia article on relations in databases article, we find the following:

> In relational database theory, a relation, as originally defined by E. F. Codd,[1] is a set of tuples (d1, d2, ..., dn), where each element dj is a member of Dj, a data domain. Codd's original definition notwithstanding, and contrary to the usual definition in mathematics, there is no ordering to the elements of the tuples of a relation.[2][3] Instead, each element is termed an attribute value. An attribute is a name paired with a domain (nowadays more commonly referred to as a type or data type). An attribute value is an attribute name paired with an element of that attribute's domain, and a tuple is a set of attribute

> values in which no two distinct elements have the same name. Thus, in some accounts, a tuple is described as a function, mapping names to values.

In a relational database, we deal with relations. The main property of a relation is that all the tuples that belong to a relation share a common data definition: they have the same list of attributes, and each attribute is of a specific data type. Then we might also might have some more constraints.

In this chapter, we are going to see what data types PostgreSQL makes available to us as application developers, and how to use them to enhance our application correctness, succinctness and performance.

Serialization and Deserialization

It's all too common to see *RDBMS* mentioned as a solution to marshaling and unmarshaling in-memory objects, and even distributed computed systems tend to talk about the *storage* parts for databases. In my opinion, we should talk about *transactional* systems rather than *storage* when we want to talk about RDBMS and other transaction technologies. That said, *storage* is a good name for distributed file systems.

On this topic, it might be interesting to realize how Lisp introduced *print readably*. In Lisp rather than working with a compiler and then running static binary files, you work with an interactive *REPL* where the *reader* and the *printer* are fully specified parts of the system. Those pieces are meant to be used by Lisp users. Here's what the common Lisp standard documentation has to say about printing *readably*:

> If *print-readably* is true, some special rules for printing objects go into effect. Specifically, printing any object O1 produces a printed representation that, when seen by the Lisp reader while the standard readtable is in effect, will produce an object O2 that is similar to O1.

In the following example code, we define a structure with *slots* of different types: string, float, and integer. Then we create an instance of that structure, with specific values for the three slots, and serialize this instance to string, only to read it back from the string:

```
(defpackage #:readably
  (:use #:cl))

(in-package #:readably)

(defstruct foo
  (name nil :type (or nil string))
  (x    0.0 :type float)
  (n    0   :type fixnum))

(defun print-and-read ()
  (let ((instance (make-foo :name "bar" :x 1.0 :n 2)))
    (values instance
            (read-from-string
             (write-to-string instance :escape t :readably t)))))
```

The result is, as expected, a couple of very similar instances:

```
CL-USER> (readably::print-and-read)
#S(READABLY::FOO :NAME "bar" :X 1.0 :N 2)
#S(READABLY::FOO :NAME "bar" :X 1.0 :N 2)
```

The first instance is created in the application code from literal strings and numbers, and the second instance has been created by the reader from a string, which could have been read from a file or a network service somewhere.

The discovery of Lisp predates the invention of the relational model by a long shot, and Lisp wasn't unique in its capacity to read data structure in-memory from *external* storage.

It is important to understand which problem can be solved with using a database service, and to insist that storing and retrieving values out of and back into memory isn't a problem for which you need a database system.

Some Relational Theory

Back to relational database management systems and what they can provide to your application is:

- A service to access your data and run transactions
- A common API to guarantee consistency in between saeveral application bases
- A transport mechanism to exchange data with the database service.

In this chapter, the focus is the C of *ACID*, i.e. data *consistency*. When your application grows, it's going to be composed of several parts: the administration panel, the customer back-office application, the public front of the application, the accounting reports, financial reporting, and maybe some more parts such as salespeople back-office and the like. Maybe some of those elements are going to be implemented using a third-party solution. Even if it's all in-house, it's often the case that different technical stacks are going to be used for different parts: a backend in Go or in Java, a frontend in Python (Django) or Ruby (on Rails), maybe PHP or Node.js, etc.

For this host of applications to work well together and respect the same set of business rules, we need a core system that enables to guaranteeing overall *consistency*. That is the main problem that a *relational database management system* is meant to solve, and that's why the relational model is so generic.

In the next chapter — Data Modeling — we are going to compare *schema-less* with the relational modeling and go more deeply into this topic. In order to be able to compare those very different approaches, we need a better understanding of how the *consistency* is guaranteed by our favorite database system, PostgreSQL.

Attribute Values, Data Domains and Data Types

The Wikipedia definition for *relation* mentions *attribute values* that are part of *data domains*. A domain here is much like in mathematics, a set of values that are given a common name to. There's the data domain of natural numbers, and the data domain of rational numbers, in mathematics.

In relational theory, we can compose basic data domains into a tuple. Allow me to quote Wikipedia again, this time the tuple definition page:

> *The term originated as an abstraction of the sequence: single, double, triple, quadruple, quintuple, sextuple, septuple, octuple, . . . , ntuple, . . . , where the prefixes are taken from the Latin names of the numerals.*

So by definition, a tuple is a list of T attributes, and a relation is a list of tuples that all share the same list of attributes domains: names and data

type.

So the basics of the relational model is to establish consistency within your data set: we structure the data in a way that we know what we are dealing with, and in a way allowing us to enforce business constraints.

The first business constraint enforced here is dealing with proper data. For instance, the *timestamp* data type in PostgreSQL implements the Gregorian Calendar, in which there's no year zero, or month zero, or day zero. While other systems might accept "timestamp formatted" text as an attribute value, PostgreSQL actually checks that the value makes sense within the Gregorian Calendar:

```
1 select date '2010-02-29';

ERROR:  date/time field value out of range: "2010-02-29"
LINE 1: select date '2010-02-29';
                    ^
```

The year 2010 isn't a leap year in the Gregorian Calendar, thus the 29th of February 2010 is not a proper date, and PostgreSQL knows that. By the way, this input syntax is named a *decorated literal*: we decorate the literal with its data type so that PostgreSQL doesn't have to guess what it is.

Let's try the infamous zero-timestamp:

```
1 select timestamp '0000-00-00 00:00:00';

ERROR:  date/time field value out of range: "0000-00-00 00:00:00"
```

No luck, because the Gregorian Calendar doesn't have a year zero. The year 1 BC is followed by 1 AD, as we can see here:

```
1 select date(date '0001-01-01' + x * interval '1 day')
2   from generate_series (-2, 1) as t(x);

     date
═══════════════
 0001-12-30 BC
 0001-12-31 BC
 0001-01-01
 0001-01-02
(4 rows)
```

We can see in the previous example that implementing the Gregorian calendar is not a restriction to live with, rather it's a powerful choice that we can put to good use. PostgreSQL knows all about leap years and time

zones, and its *time* and *date* data types also implement nice support for meaningful values:

```
select date 'today' + time 'allballs' as midnight;
```

```
      midnight
═══════════════════════
 2017-08-14 00:00:00
(1 row)
```

The *allballs* time literal sounds like an Easter egg — its history is explained in this pgsql-docs thread.

Consistency and Data Type Behavior

A key aspect of PostgreSQL data types lies in their behavior. Comparable to an *object-oriented* system, PostgreSQL implements functions and operator polymorphism, allowing for the dispatching of code at run-time depending on the types of arguments.

If we have a closer look at a very simple SQL query, we can see lots happening under the hood:

```
select code from drivers where driverid = 1;
```

In this query, the expression *driverid = 1* uses the = operator in between a column name and a literal value. PostgreSQL knows from its catalogs that the *driverid* column is a *bigint* and parses the literal *1* as an integer. We can check that with the following query:

```
select pg_typeof(driverid), pg_typeof(1) from drivers limit 1;
```

```
 pg_typeof │ pg_typeof
═══════════╪═══════════
 bigint    │ integer
(1 row)
```

Now, how does PostgreSQL implements = in between an 8 bytes integer and a 4 bytes integer? Well it turns out that this decision is dynamic: the operator = dispatches to an established function depending on the types of its left and right operands. We can even have a look at the PostgreSQL catalogs to get a better grasp of this notion:

```
  select oprname, oprleft::regtype, oprcode::regproc
    from pg_operator
   where oprname = '='
```

```
4 |     and oprleft::regtype::text ~ 'int|time|text|circle|ip'
5 | order by oprleft;
```

This gives us a list of the following instances of the = operator:

```
 oprname |           oprleft            |         oprcode
═════════╪══════════════════════════════╪═══════════════════════════
 =       | bigint                       | int84eq
 =       | bigint                       | int8eq
 =       | bigint                       | int82eq
 =       | smallint                     | int28eq
 =       | smallint                     | int2eq
 =       | smallint                     | int24eq
 =       | int2vector                   | int2vectoreq
 =       | integer                      | int48eq
 =       | integer                      | int42eq
 =       | integer                      | int4eq
 =       | text                         | texteq
 =       | abstime                      | abstimeeq
 =       | reltime                      | reltimeeq
 =       | tinterval                    | tintervaleq
 =       | circle                       | circle_eq
 =       | time without time zone       | time_eq
 =       | timestamp without time zone  | timestamp_eq
 =       | timestamp without time zone  | timestamp_eq_date
 =       | timestamp without time zone  | timestamp_eq_timestamptz
 =       | timestamp with time zone     | timestamptz_eq_timestamp
 =       | timestamp with time zone     | timestamptz_eq
 =       | timestamp with time zone     | timestamptz_eq_date
 =       | interval                     | interval_eq
 =       | time with time zone          | timetz_eq
(24 rows)
```

The previous query limits its output to the datatype expected on the *left* of the operator. Of course, the catalogs also store the datatype expected on the *right* of it, and the result type too, which is *Boolean* in the case of equality. The *oprcode* column in the output is the name of the PostgreSQL function that is run when the operator is used.

In our case with *driverid = 1* PostgreSQL is going to use the *int84eq* function to implement our query. This is true unless there's an index on *driverid* of course, in which case PostgreSQL will walk the index to find matching rows without comparing the literal with the table's content, only with the index content.

When using PostgreSQL, data types provide the following:

- Input data representation, expected in input literal values

- Output data representation
- A set of functions working with the data type
- Specific implementations of existing functions for the new data type
- Operator specific implementations for the data type
- Indexing support for the data type

The indexing support for PostgreSQL covers several kinds of indexes: *B-tree*, *GiST*, *GIN*, *SP-GiST*, *hash* and *brin*. This book doesn't fo further and cover the details of each of those index types. As an example of data type support for some indexes and the relationship in between a data type, a support function, an operator and an index, we can have a look at the *GiST* support for the *ip4r* extension data type:

```
select amopopr::regoperator
    from pg_opclass c
         join pg_am am on am.oid = c.opcmethod
         join pg_amop amop on amop.amopfamily = c.opcfamily
   where opcintype = 'ip4r'::regtype
     and am.amname = 'gist';
```

The *pg_opclass* catalog is a list of *operator class*, each of them belongs to an *operator family* as found in the *pg_opfamily* catalog. Each index type implements an *access method* represented in the *pg_am* catalog. Finally, each operator that may be used in relation to an index access method is listed in the *pg_amop* catalog.

Knowing that we can access the PostgreSQL catalogs at run-time and discover the *ip4r* supported operators for a *GiST* indexed lookup:

```
    amopopr
═══════════════
 >>=(ip4r,ip4r)
 <<=(ip4r,ip4r)
 >>(ip4r,ip4r)
 <<(ip4r,ip4r)
 &&(ip4r,ip4r)
 =(ip4r,ip4r)
(6 rows)
```

Those catalog queries are pretty advanced material that you don't need in your daily life as an application developer. That said, it's good to have some understanding of how things work in PostgreSQL as it allows a smarter usage of the system you are already relying on for your data.

What we've seen here is that PostgreSQL implementation of data types is a completely dynamic system with function and operator dispatch, and PostgreSQL extension authors have APIs they can use to register new

indexing support at run time (when you type in *create extension*).

The goal of understanding that is for you, as an application developer, to understand how much can be done in PostgreSQL thanks to the integral concept of data type.

PostgreSQL Data Types

PostgreSQL comes with a long list of data types. The following query limits the types to the ones directly interesting to someone who is an application developer, and still it lists 72 data types:

```
select nspname, typname
    from      pg_type t
         join pg_namespace n
           on n.oid = t.typnamespace
   where nspname = 'pg_catalog'
     and typname !~ '(^_|^pg_|^reg|_handler$)'
order by nspname, typname;
```

Let's take only a sample of those with the help of the *TABLESAMPLE* feature of PostgreSQL, documented in the select SQL from page of the documentation:

```
select nspname, typname
    from      pg_type t TABLESAMPLE bernoulli(20)
         join pg_namespace n
           on n.oid = t.typnamespace
   where nspname = 'pg_catalog'
     and typname !~ '(^_|^pg_|^reg|_handler$)'
order by nspname, typname;
```

In this run here's what I get as a random sample of about 20% of the available PostgreSQL types. If you run the same query again, you will have a different result set:

```
  nspname   |    typname
════════════╪═══════════════
 pg_catalog | abstime
 pg_catalog | anyelement
 pg_catalog | bool
 pg_catalog | cid
 pg_catalog | circle
 pg_catalog | date
 pg_catalog | event_trigger
```

```
 pg_catalog | line
 pg_catalog | macaddr
 pg_catalog | oidvector
 pg_catalog | polygon
 pg_catalog | record
 pg_catalog | timestamptz
(13 rows)
```

Our pick for the data types in this book isn't based on a *table sample* query, though. Yes, it would be some kind of fun to do it like this, but maybe not the kind you're expecting from the pages of this book...

Boolean

The Boolean data type has been the topic of the three valued logic section earlier in this book, with the SQL boolean truth table that includes the values *true*, *false* and *null*, and it's important enough to warrant another inclusion here:

```
   a    |   b   |  a=b  |      op       |  result
════════╪═══════╪═══════╪═══════════════╪═══════════
 true   | true  | true  | true = true   | is true
 true   | false | false | true = false  | is false
 true   | ¤     | ¤     | true = null   | is null
 false  | true  | false | false = true  | is false
 false  | false | true  | false = false | is true
 false  | ¤     | ¤     | false = null  | is null
 ¤      | true  | ¤     | null = true   | is null
 ¤      | false | ¤     | null = false  | is null
 ¤      | ¤     | ¤     | null = null   | is null
(9 rows)
```

You can have tuple attributes as Booleans too, and PostgreSQL includes specific aggregates for them:

```
  select year,
         format('%s %s', forename, surname) as name,
         count(*) as ran,
         count(*) filter(where position = 1) as won,
         count(*) filter(where position is not null) as finished,
         sum(points) as points
    from      races
         join results using(raceid)
         join drivers using(driverid)
group by year, drivers.driverid
  having bool_and(position = 1) is true
order by year, points desc;
```

In this query, we show the *bool_and()* aggregates that returns true when all the Boolean input values are true. Like every *aggregate* it silently bypasses *null* by default, so in our expression of *bool_and(position = 1)* we will filter F1 drivers who won all the races they finished in a specific season:

```
 year │        name         │ ran │ won │ finished │ points
══════╪═════════════════════╪═════╪═════╪══════════╪════════
 1950 │ Juan Fangio         │   7 │   3 │        3 │     27
 1950 │ Johnnie Parsons     │   1 │   1 │        1 │      9
 1951 │ Lee Wallard         │   1 │   1 │        1 │      9
 1952 │ Alberto Ascari      │   7 │   6 │        6 │   53.5
 1952 │ Troy Ruttman        │   1 │   1 │        1 │      8
 1953 │ Bill Vukovich       │   1 │   1 │        1 │      9
 1954 │ Bill Vukovich       │   1 │   1 │        1 │      8
 1955 │ Bob Sweikert        │   1 │   1 │        1 │      8
 1956 │ Pat Flaherty        │   1 │   1 │        1 │      8
 1956 │ Luigi Musso         │   4 │   1 │        1 │      5
 1957 │ Sam Hanks           │   1 │   1 │        1 │      8
 1958 │ Jimmy Bryan         │   1 │   1 │        1 │      8
 1959 │ Rodger Ward         │   2 │   1 │        1 │      8
 1960 │ Jim Rathmann        │   1 │   1 │        1 │      8
 1961 │ Giancarlo Baghetti  │   3 │   1 │        1 │      9
 1966 │ Ludovico Scarfiotti │   2 │   1 │        1 │      9
 1968 │ Jim Clark           │   1 │   1 │        1 │      9
(17 rows)
```

If we want to restrict the results to drivers who finished *and* won every race they entered in a season we need to then write *having bool_and(position is not distinct from 1) is true*, and then the result set only contains those drivers who participated in a single race in the season.

The main thing about Booleans is the set of operators to use with them:

- The = doesn't work as you think it would
- Use *is* to test against literal *true*, *false* or *null* rather than =
- Remember to use the *is distinct from* and *is not distinct from* operators when you need them,
- Booleans can be aggregated thanks to *bool_and* and *bool_or*.

The main thing about Booleans in SQL is that they have three possible values: *true*, *false* and *null*. Moreover the behavior with *null* is entirely ad-hoc, so either you remember it or you remember to check your assumptions. For more about this topic, you can read What is the deal with NULLs? from PostgreSQL Contributor Jeff Davis.

Character and Text

PostgreSQL knows how to deal with characters and text, and it implements several data types for that, all documented in the character types chapter of the documentation.

About the data type itself, it must be noted that *text* and *varchar* are the same thing as far as PostgreSQL is concerned, and *character varying* is an alias for *varchar*. When using *varchar(15)* you're basically telling PostgreSQL to manage a *text* column with a *check* constraint of 15 characters.

Yes PostgreSQL knows how to count characters even with Unicode encoding, more on that later.

There's a very rich set of PostgreSQL functions to process text — you can find them all in the string functions and operators documentation chapter — with functions such as *overlay()*, *substring()*, *position()* or *trim()*. Or aggregates such as *string_agg()*. There are also *regular expression* functions, including the very powerful *regexp_split_to_table()*.

For more about PostgreSQL regular expressions, read the main documentation about them in the pattern matching chapter.

Additionnaly to the classic *like* and *ilike* patterns and to the SQL standard *similar to* operators, PostgreSQL embeds support for a full-blown *regular expression* matching engine. The main operator implementing regexp is `~`, and then you find the derivatives for *not matching* and *match either case*. In total, we have four operators: `~`, `!~`, `~*` and `!~*`.

Note that PostgreSQL also supports indexing for regular expressions thanks to its trigram extension: pg_trgm.

The *regular expression* split functions are powerful in many use cases. In particular, they are very helpful when you have to work with a messy schema, in which a single column represents several bits of information in a pseudo specified way. An example of such a dataset is available in open data: the Archives de la Planète or "planet archives". The data is available as CSV and once loaded looks like this:

```
\pset format wrapped
\pset columns 70
table opendata.archives_planete limit 1;
```

And we get the following sample data, all in French (but it doesn't matter very much for our purposes here):

```
-[ RECORD 1 ]-------------------------------------------
id          | IF39599
inventory   | A 2 037
orig_legend | Serbie, Monastir Bitolj, Un Turc
legend      | Un Turc
location    | Monastir (actuelle Bitola), Macédoine
date        | mai 1913
operator    | Auguste Léon
...
themes      | Habillement > Habillement traditionnel,Etres ⋯
            |⋯humains > Homme,Etres humains > Portrait,Rela⋯
            |⋯tions internationales > Présence étrangère
...
collection  | Archives de la Planète
...
```

You can see that the *themes* column contains several categories for a single entry, separated with a comma. Within that comma separated list, we find another classification, this time separated with a greater than sign, which looks like a hierarchical categorization of the themes.

So this picture id *IF39599* actually is relevant to that series of themes:

```
   id    |           cat1            |           cat2
═════════╪═══════════════════════════╪═══════════════════════════
 IF39599 | Habillement               | Habillement traditionnel
 IF39599 | Etres humains             | Homme
 IF39599 | Etres humains             | Portrait
 IF39599 | Relations internationales | Présence étrangère
(4 rows)
```

The question is, how do we get that information? Also, is it possible to have an idea of the distribution of the whole data set in relation to the categories embedded in the *themes* column?

With PostgreSQL, this is easy enough to achieve. First, we are going to split the *themes* column using a regular expression:

```
1 select id, regexp_split_to_table(themes, ',')
2   from opendata.archives_planete
3  where id = 'IF39599';
```

We get the following table:

```
   id    |             regexp_split_to_table
═════════╪═══════════════════════════════════════════════
 IF39599 | Habillement > Habillement traditionnel
 IF39599 | Etres humains > Homme
```

```
 IF39599 | Etres humains > Portrait
 IF39599 | Relations internationales > Présence étrangère
(4 rows)
```

Now that we have a table with an entry per theme for the same document, we can further split each entry into the two-levels category that it looks like. We do that this time with *regexp_split_to_array()* so as to keep the categories together:

```
select id,
       regexp_split_to_array(
         regexp_split_to_table(themes, ','),
         ' > ')
       as categories
  from opendata.archives_planete
 where id = 'IF39599';
```

And now we have:

```
   id    |                      categories
═════════╪═══════════════════════════════════════════════════
 IF39599 | {Habillement,"Habillement traditionnel"}
 IF39599 | {"Etres humains",Homme}
 IF39599 | {"Etres humains",Portrait}
 IF39599 | {"Relations internationales","Présence étrangère"}
(4 rows)
```

We're almost there. For the content to be normalized we want to have the categories in their own separate columns, say *category* and *subcategory*:

```
with categories(id, categories) as
 (
   select id,
          regexp_split_to_array(
            regexp_split_to_table(themes, ','),
            ' > ')
          as categories
     from opendata.archives_planete
 )
 select id,
        categories[1] as category,
        categories[2] as subcategory
   from categories
  where id = 'IF39599';
```

And now we make sense of the open data:

```
   id    |         category          |       subcategory
═════════╪═══════════════════════════╪═══════════════════════════
 IF39599 | Habillement               | Habillement traditionnel
 IF39599 | Etres humains             | Homme
```

```
 IF39599 | Etres humains             | Portrait
 IF39599 | Relations internationales | Présence étrangère
(4 rows)
```

As a side note, cleaning up a data set after you've imported it into PostgreSQL makes the difference clear between the classic *ETL* jobs (extract, transform, load) and the powerful *ELT* jobs (extract, load, transform) where you can transform your data using a data processing language: SQL.

So, now that we know how to have a clear view of the dataset, let's inquire about the categories used in our dataset:

```
with categories(id, categories) as
  (
    select id,
           regexp_split_to_array(
             regexp_split_to_table(themes, ','),
             ' > ')
           as categories
      from opendata.archives_planete
  )
  select categories[1] as category,
         categories[2] as subcategory,
         count(*)
    from categories
group by rollup(category, subcategory);
```

That query returns 175 rows, so here's an extract only:

```
       category          |         subcategory          | count
═══════════════════════╪══════════════════════════════╪═══════
 Activite économique     | Agriculture / élevage        |   138
 Activite économique     | Artisanat                    |    81
 Activite économique     | Banque / finances            |     2
 Activite économique     | Boutique / magasin           |    39
 Activite économique     | Commerce ambulant            |     5
 Activite économique     | Commerce extérieur           |     1
 Activite économique     | Cueillette / chasse          |     9
...
 Art                     | Peinture                     |    15
 Art                     | Renaissance                  |    52
 Art                     | Sculpture                    |    87
 Art                     | Théâtre                      |     7
 Art                     | ¤                            |   333
...
 Habillement             | Uniforme scolaire            |     1
 Habillement             | Vêtement de travail          |     3
 Habillement             | ¤                            |   163
 Habitat / Architecture  | Architecture civile publique |    37
 Habitat / Architecture  | Architecture commerciale     |    24
 Habitat / Architecture  | Architecture de jardin       |    31
```

```
...
 Vie quotidienne        | Vie domestique                |     8
 Vie quotidienne        | Vie rurale                    |     5
 Vie quotidienne        | ¤                             |    64
 ¤                      | ¤                             |  4449
(175 rows)
```

Each *subcategory* appears only within the same *category* each time, and we've chosen to do a *roll up* analysis of our data set here. Other *grouping sets* are available, such as the *cube*, or manually editing the dimensions you're interested into.

In an *ELT* assignment, we would create a new *categories* table containing each entry we saw in the rollup query only once, as a catalog, and an association table in between the main *opendata.archives_planete* table and this categories catalog, where each archive entry might have several categories and subcategories assigned and each category, of course, might have several archive entries assigned.

Here, the topic is about text function processing in PostgreSQL, so we just run the query against the base data set.

Finally, when mentioning advanced string matching and the *regular expression*, we must also mention PostgreSQL's implementation of a full text search with support for *documents*, advanced *text search queries*, *ranking*, *highlighting*, *pluggable parsers*, *dictionaries* and *stemmers*, *synonyms*, and *thesaurus*. Additionally, it's possible to configure all those pieces. This is material for another book, so if you need advanced searches of documents that you manage in PostgreSQL please read the documentation about it. There are also many online resources on the topic too.

Server Encoding and Client Encoding

When addressing the text datatype we must mention encoding settings, and possibly also issues. An encoding is a particular representation of characters in bits and bytes. In the *ASCII* encoding the letter `A` is encoded as the 7-bits byte `1000001`, or 65 in decimal, or 41 in hexadecimal. All those numbers are going to be written the same way on-disk, and the letter `A` too.

The *SQL_ASCII* encoding is a trap you need to avoid falling into. To know which encoding your database is using, run the *psql* command `\l`:

```
                                  List of databases
   Name    |  Owner   | Encoding |   Collate   |    Ctype    | ···
═══════════╪══════════╪══════════╪═════════════╪═════════════╪═══
 chinook   | dim      | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
 f1db      | dim      | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
 pgloader  | dim      | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
 template0 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
 template1 | postgres | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
 yesql     | dim      | UTF8     | en_US.UTF-8 | en_US.UTF-8 | ···
(6 rows)
```

In this output, I've stripped down the last column of output for better integration for the page size here, so you don't get to see the *Access privileges* for those databases.

The encoding here is *UTF8* which is the best choice these days, and you can see that the collation and ctype are English based in the *UTF-8* encoding, which is good for my installation. You might, of course, pick something else.

The non-encoding *SQL_ASCII* accepts any data you throw at it, whereas the *UTF8* encoding (and some others) do check for valid input. Never use *SQL_ASCII*, as you will not be able to retrieve data in any encoding and will lose data because of that! Migrating away from *SQL_ASCII* to a proper encoding such as *UTF8* is possible byt lossy and complex.

You can also have an *UTF8* encoded database and use a legacy application (or programming language) that doesn't know how to handle Unicode properly. In that case, you can ask PostgreSQL to convert all and any data on the fly between the server-side encoding and your favorite client-side encoding, thanks to the *client_encoding* setting.

```
1 │show client_encoding;
```

Here again, we use UTF8 client side, which allows handling French accentuated characters we saw previously.

```
 client_encoding
═════════════════
 UTF8
(1 row)
```

Be aware that not all combinations of *server encoding* and *client encoding* make sense. While it is possible for PostgreSQL to communicate with your application using the *latin1* encoding on the client side if the server side dataset includes texts in incompatible encodings, PostgreSQL will issue an

error. Such texts might be written using non-Latin scripts such as Cyrillic, Chinese, Japanese, Arabic or other languages.

From the Emacs facility `M-x view-hello-file`, here's a table with spelling of hello in plenty of different languages and scripts, all encoded in *UTF8*:

language	hello
Czech (čeština)	Dobrý den
Danish (dansk)	Hej / Goddag / Halløj
Dutch (Nederlands)	Hallo / Dag
English /'ɪŋglɪʃ/	Hello
Esperanto	Saluton (Eĥoŝanĝo ĉiuĵaŭde)
Estonian (eesti keel)	Tere päevast / Tere õhtust
Finnish (suomi)	Hei / Hyvää päivää
French (français)	Bonjour / Salut
Georgian (ქართველი)	გამარჯობა
German (Deutsch)	Guten Tag / Grüß Gott
Greek (ελληνικά)	Γειά σας
Greek, ancient (ἑλληνική)	Οὗλέ τε καὶ μέγα χαῖρε
Hungarian (magyar)	Szép jó napot!
Italian (italiano)	Ciao / Buon giorno
Maltese (il-Malti)	Bonġu / Saħħa
Mathematics	∀ p ∈ world • hello p □
Mongolian (монгол хэл)	Сайн байна уу?
Norwegian (norsk)	Hei / God dag
Polish (język polski)	Dzień dobry! / Cześć!
Russian (русский)	Здра́вствуйте!
Slovak (slovenčina)	Dobrý deň
Slovenian (slovenščina)	Pozdravljeni!
Spanish (español)	¡Hola!
Swedish (svenska)	Hej / Goddag / Hallå
Turkish (Türkçe)	Merhaba
Ukrainian (українська)	Вітаю
Vietnamese (ti□ ng Việt)	Chào bạn
Japanese (日本語)	こんにちは / □□□□□
Chinese (中文,普通话,汉语)	你好
Cantonese (粵語,廣東話)	早晨, 你好

Now, of course, I can't have that data sent to me in *latin1*:

```
yesql# set client_encoding to latin1;
SET
yesql# select * from hello where language ~ 'Georgian';
ERROR:  character with byte sequence 0xe1 0x83 0xa5 in encoding "UTF8" ⏎
has no equivalent in encoding "LATIN1"
yesql# reset client_encoding ;
RESET
```

So if it's possible for you, use *UTF-8* encoding and you'll have a much simpler life. It must be noted that Unicode encoding makes comparing

and sorting text a rather costly operation. That said being fast and wrong is not an option, so we are going to still use unicode text!

Numbers

PostgreSQL implement multiple data types to handle numbers, as seen in the documentation chapter about numeric types:

- *integer*, 32 bits signed numbers
- *bigint*, 64 bits signed numbers
- *smallint*, 16 bits signed numbers
- *numeric*, arbitrary precision numbers
- *real*, 32 bits floating point numbers with 6 decimal digits precision
- *double precision*, 64 bits floating point numbers with 15 decimal digits precision

We mentioned before that the SQL query system is statically typed, and PostgreSQL must establish the data type of every column of a query input and result-set before being able to plan and execute it. For numbers, it means that the type of every number literal must be derived at query parsing time.

In the following query, we count how many times a driver won a race when he started in pole position, per season, and return the ten drivers having done that the most in all the records or Formula One results. The query uses integer expressions *grid = 1* and *position = 1* and PostgreSQL is left to figure out which data type does that literal value 1 belong to?

It could be an *smallint*, an *integer* or a *bigint*. It could also be a *numeric* value. Of course knowing that the *grid* and *position* columns are of type *bigint* might have an impact on the parsing choice here.

```
  select year,
         drivers.code,
         format('%s %s', forename, surname) as name,
         count(*)
    from results
         join races using(raceid)
         join drivers using(driverid)
   where grid = 1
     and position = 1
group by year, drivers.driverid
```

```
order by count desc
   limit 10;
```

Which by the way gives:

```
 year | code |        name         | count
══════╪══════╪═════════════════════╪═══════
 1992 | ¤    | Nigel Mansell       |     9
 2011 | VET  | Sebastian Vettel    |     9
 2013 | VET  | Sebastian Vettel    |     8
 2004 | MSC  | Michael Schumacher  |     8
 2016 | HAM  | Lewis Hamilton      |     7
 2015 | HAM  | Lewis Hamilton      |     7
 1988 | ¤    | Ayrton Senna        |     7
 1991 | ¤    | Ayrton Senna        |     7
 2001 | MSC  | Michael Schumacher  |     6
 2014 | HAM  | Lewis Hamilton      |     6
(10 rows)
```

Also impacting on the PostgreSQL parsing choice of a data type for the 1 literal is the = operator, which exists in three different variants when its left operand is a *bigint* value:

```
  select oprname,
         oprcode::regproc,
         oprleft::regtype,
         oprright::regtype,
         oprresult::regtype
    from pg_operator
   where oprname = '='
     and oprleft::regtype = 'bigint'::regtype;
```

We can see that PostgreSQL must support the = operator for every possible combination of its integer data types:

```
 oprname | oprcode | oprleft | oprright | oprresult
═════════╪═════════╪═════════╪══════════╪═══════════
 =       | int8eq  | bigint  | bigint   | boolean
 =       | int84eq | bigint  | integer  | boolean
 =       | int82eq | bigint  | smallint | boolean
(3 rows)
```

Short of that, we would have to use decorated literals for numbers in all our queries, writing:

```
   where grid = bigint '1' and position = bigint '1'
```

The combinatorial explosion of internal operators and support functions for comparing numbers is the reason why the PostgreSQL project has chosen to have a minimum number of numeric data types: the impacts of adding

another one is huge in terms of query planning time and internal data structure sizing. That's why there are no *unsigned* numeric data types in PostgreSQL.

Floating Point Numbers

Adding to integer data type support, PostgreSQL also has support for floating point numbers. Please take some time to read What Every Programmer Should Know About Floating-Point Arithmetic before considering any serious use of floating point numbers. In short, there are some numbers that can't be represented in base 10, such as 1/3. In base 2 also, some numbers are not possible to represent, and it's a different set than in base 10. So in base 2, you can't possibly represent 1/5 or 1/10, for example.

In short, understand what you're doing when using *real* or *double precision* data types, and never use them to deal with money. Use either *numeric* which provides arbitrary precision or an *integer* based representation of the money.

Sequences and the Serial Pseudo Data Type

Other kinds of numeric data types in PostgreSQL are the *smallserial*, *serial* and *bigserial* data types. They actually are *pseudo types*: the parser recognize their syntax, but then transforms them into something else entirely. Straight from the excellent PostgreSQL documentation again:

```
CREATE TABLE tablename (
    colname SERIAL
);
```

This is equivalent to specifying:

```
CREATE SEQUENCE tablename_colname_seq;
CREATE TABLE tablename (
    colname integer NOT NULL DEFAULT nextval('tablename_colname_seq')
);
ALTER SEQUENCE tablename_colname_seq OWNED BY tablename.colname;
```

The *sequence* SQL object is covered by the SQL standard and documented in the create sequence manual entry for PostgreSQL. This object is the only one in SQL with a non-transactional behavior. Of course, that's

on purpose, so that multiple sessions can get the next number from the sequence concurrently, without having to then wait until *commit;* to decide if they can keep their sequence number.

From the docs:

> Sequences are based on bigint arithmetic, so the range cannot exceed the range of an eight-byte integer (-9223372036854775808 to 9223372036854775807).

So if you have a *serial* column, its real type is going to be *integer*, and as soon as the sequence generates a number that doesn't fit into signed 4-byte representation, you're going to have errors.

In the following example, we construct the situation in which we exhaust the *id* column (an *integer*) and still use the sequence to generate the next entry:

```
create table seq(id serial);
CREATE TABLE

select setval('public.seq_id_seq'::regclass, 2147483647);
   setval
════════════
 2147483647
(1 row)

yesql# insert into public.seq values (default);
ERROR:  integer out of range
```

That could happen to your application while in production if you use *serial* rather than *bigserial*. If you need a sequence and need to restrict your column to 4-byte integers, then you need to implement a maintenance policy around the fact that the sequence is 8 bytes and the hosting column only 4.

Universally Unique Identifier: UUID

A universally unique identifier (*UUID*) is a 128-bit number used to identify information in computer systems. The term globally unique identifier (GUID) is also used. PostgreSQL implements support for UUID, both for storing and processing them, and also with the *uuid-ossp* extension, for generating them.

If you need to generate UUIDs from PostgreSQL, which we do in order to cover the topic in this book, then install the extension. The extension is part of the PostgreSQL contribs, so you need to have that OS package installed.

```
create extension "uuid-ossp";
```

Now we can have a look at those UUIDs:

```
select uuid_generate_v4()
  from generate_series(1, 10) as t(x);
```

Here's a list of locally generated UUID v4:

```
           uuid_generate_v4
═══════════════════════════════════════
 fbb850cc-dd26-4904-96ef-15ad8dfaff07
 0ab19b19-c407-410d-8684-1c3c7f978f49
 5f401a04-2c58-4cb1-b203-ae2b2a1a4a5e
 d5043405-7c03-40b1-bc71-aa1e15e1bbf4
 33c98c8a-a24b-4a04-807f-33803faa5f0a
 c68b46eb-b94f-4b74-aecf-2719516994b7
 5bf5ec69-cdbf-4bd1-a533-7e0eb266f709
 77660621-7a9b-4e59-a93a-2b33977e84a7
 881dc4f4-b587-4592-a720-81d9c7e15c63
 1e879ef4-6f1f-4835-878a-8800d5e9d4e0
(10 rows)
```

Even if you generate UUIDs from your application, managing them as a proper UUID in PostgreSQL is a good idea, as PostgreSQL actually stores the binary value of the UUID on 128 bits (or 16 bytes) rather than way more when storing the text representation of an UUID:

```
select pg_column_size(uuid 'fbb850cc-dd26-4904-96ef-15ad8dfaff07')
       as uuid_bytes,

       pg_column_size('fbb850cc-dd26-4904-96ef-15ad8dfaff07')
       as uuid_string_bytes;
```

```
 uuid_bytes │ uuid_string_bytes
════════════╪═══════════════════
         16 │                37
(1 row)
```

Should we use UUIDs as identifiers in our database schemas? We get back to that question in the next chapter.

Bytea and Bitstring

PostgreSQL can store and process raw binary values, which is sometimes useful. Binary columns are limited to about 1 GB in size (8 bytes of this are used in the header out of this). Those types are documented in the PostgreSQL chapter entitled Binary Data Types.

While it's possible to store large binary data that way, PostgreSQL doesn't implement a chunk API and will systematically fetch the whole content when the column is included in your queries output. That means loading the content from disk to memory, pushing it through the network and handling it as a whole in-memory on the client-side, so it's not always the best solution around.

That said, when storing binary content in PostgreSQL it is then automatically part of your online backups and recovery solution, and the online backups are transactional. So if you need to have binary content with transactional properties, *bytea* might be exactly what you need.

Date/Time and Time Zones

Handling dates and time and time zones is a very complex matter, and on this topic, you can read Erik Naggum's piece The Long, Painful History of Time.

The PostgreSQL documentation chapters with the titles Date/Time Types, Data Type Formatting Functions, and Date/Time Functions and Operators cover all you need to know about date, time, timestamps, and time zones with PostgreSQL.

The first question we need to answer here is about using timestamps with or without *time zones* from our applications. The answer is simple: always use *timestamps WITH time zones*.

A common myth is that storing time zones will certainly add to your storage and memory footprint. It's actually not the case:

```
1 select pg_column_size(timestamp without time zone 'now'),
2        pg_column_size(timestamp with time zone 'now');
```

```
 pg_column_size | pg_column_size
════════════════╪════════════════
```

```
               8 |               8
(1 row)
```

PostgreSQL defaults to using *bigint* internally to store timestamps, and the on-disk and in-memory format are the same with or without time zone support. Here's their whole type definition in the PostgreSQL source code (in `src/include/datatype/timestamp.h`):

```
typedef int64 Timestamp;
typedef int64 TimestampTz;
```

From the PostgreSQL documentation for timestamps, here's how it works:

> For timestamp with time zone, the internally stored value is always in UTC (Universal Coordinated Time, traditionally known as Greenwich Mean Time, GMT). An input value that has an explicit time zone specified is converted to UTC using the appropriate offset for that time zone. If no time zone is stated in the input string, then it is assumed to be in the time zone indicated by the system's TimeZone parameter, and is converted to UTC using the offset for the timezone zone.

PostgreSQL doesn't store the time zone they come from with your timestamp. Instead it converts to and from the input and output timezone much like we've seen for text with *client_encoding*.

```
begin;

drop table if exists tstz;

create table tstz(ts timestamp, tstz timestamptz);

set timezone to 'Europe/Paris';
select now();
insert into tstz values(now(), now());

set timezone to 'Pacific/Tahiti';
select now();
insert into tstz values(now(), now());

set timezone to 'Europe/Paris';
table tstz;

set timezone to 'Pacific/Tahiti';
table tstz;

commit;
```

In this script, we play with the client's setting *timezone* and change from a French value to another French value, as Tahiti is an island in the Pacific that is part of France. Here's the full output as seen when running this script, when launched with `psql -a -f tz.sql`:

```
BEGIN
...
set timezone to 'Europe/Paris';
SET
select now();
              now
═══════════════════════════════
 2017-08-19 14:22:11.802755+02
(1 row)

insert into tstz values(now(), now());
INSERT 0 1
set timezone to 'Pacific/Tahiti';
SET
select now();
              now
═══════════════════════════════
 2017-08-19 02:22:11.802755-10
(1 row)

insert into tstz values(now(), now());
INSERT 0 1
set timezone to 'Europe/Paris';
SET
table tstz;
             ts             │              tstz
════════════════════════════╪═══════════════════════════════
 2017-08-19 14:22:11.802755 │ 2017-08-19 14:22:11.802755+02
 2017-08-19 02:22:11.802755 │ 2017-08-19 14:22:11.802755+02
(2 rows)

set timezone to 'Pacific/Tahiti';
SET
table tstz;
             ts             │              tstz
════════════════════════════╪═══════════════════════════════
 2017-08-19 14:22:11.802755 │ 2017-08-19 02:22:11.802755-10
 2017-08-19 02:22:11.802755 │ 2017-08-19 02:22:11.802755-10
(2 rows)

commit;
COMMIT
```

First, we see that the *now()* function always returns the same timestamp within a single transaction. If you want to see the clock running while in a transaction, use the *clock_timestamp()* function instead.

Then, we see that when we change the *timezone* client setting, PostgreSQL outputs timestamps as expected, in the selected timezone. If you manage

an application with users in different time zones and you want to display time in their own local preferred time zone, then you can *set timezone* in your application code before doing any timestamp related processing, and have PostgreSQL do all the hard work for you.

Finally, when selecting back from the *tstz* table, we see that the column *tstz* realizes that both the inserted values actually are the same point in time, but seen from different places in the world, whereas the *ts* column makes it impossible to compare the entries and realize they actually happened at exactly the same time.

As said before, even when using timestamps *with* time zone, PostgreSQL will not store the time zone in use at input time, so there's no way from our *tstz* table to know that the entries are at the same time but just from different places.

The opening of this section links to The Long, Painful History of Time, and if you didn't read it yet, maybe now is a good time. Allow me to quote a relevant part of the article here:

> *The basic problem with time is that we need to express both time and place whenever we want to place some event in time and space, yet we tend to assume spatial coordinates even more than we assume temporal coordinates, and in the case of time in ordinary communication, it is simply left out entirely. Despite the existence of time zones and strange daylight saving time regimes around the world, most people are blithely unaware of their own time zone and certainly of how it relates to standard references. Most people are equally unaware that by choosing a notation that is close to the spoken or written expression of dates, they make it meaningless to people who may not share the culture, but can still read the language. It is unlikely that people will change enough to put these issues to rest, so responsible computer people need to address the issues and resist the otherwise overpowering urge to abbreviate and drop context.*

Several options are available to input timestamp values in PostgreSQL. The easiest is to use the ISO format, so if your application's code allows that you're all set. In the following example we leave the time zone out, as usually, it's handled by the *timezone* session parameter, as seen above. If you need to, of course, you can input the time zone in the timestamp

values directly:

```
select timestamptz '2017-01-08 04:05:06',
       timestamptz '2017-01-08 04:05:06+02';
```

At insert or update time, use the same literal strings without the type decoration: PostgreSQL already knows the type of the target column, and it uses that to parse the values literal in the DML statement.

Some application use-cases only need the date. Then use the *date* data type in PostgreSQL. It is of course then possible to compare a *date* and a *timestamp with time zone* in your SQL queries, and even to append a time offset on top of your date to construct a *timestamp*.

Time Intervals

PostgreSQL implements an *interval* data type along with the *time*, *date* and *timestamptz* data types. An *interval* describes a duration, like a month or two weeks, or even a millisecond:

```
set intervalstyle to postgres;

select interval '1 month',
       interval '2 weeks',
       2 * interval '1 week',
       78389 * interval '1 ms';
```

The default PostgreSQL output looks like this:

```
 interval │ interval │ ?column? │   ?column?
══════════╪══════════╪══════════╪═══════════════
 1 mon    │ 14 days  │ 14 days  │ 00:01:18.389
(1 row)
```

Several *intervalstyle* values are possible, and the setting *postgres_verbose* is quite nice for interactive *psql* sessions:

```
set intervalstyle to postgres_verbose;

select interval '1 month',
       interval '2 weeks',
       2 * interval '1 week',
       78389 * interval '1 ms';
```

This time we get a user-friendly output:

```
 interval │ interval  │ ?column?  │      ?column?
══════════╪═══════════╪═══════════╪═══════════════════
```

```
 @ 1 mon  | @ 14 days | @ 14 days | @ 1 min 18.389 secs
(1 row)
```

How long is a month? Well, it depends on which month, and PostgreSQL knows that:

```
select d::date as month,

       (d + interval '1 month' - interval '1 day')::date as month_end,

       (d + interval '1 month')::date as next_month,

       (d + interval '1 month')::date - d::date as days

  from generate_series(
                        date '2017-01-01',
                        date '2017-12-01',
                        interval '1 month'
                       )
       as t(d);
```

When you attach an *interval* to a date or timestamp in PostgreSQL then the number of days in that interval adjusts to the specific calendar entry you've picked. Otherwise, an interval of a month is considered to be 30 days. Here we see that computing the last day of February is very easy:

```
   month    | month_end  | next_month | days
════════════╪════════════╪════════════╪══════
 2017-01-01 | 2017-01-31 | 2017-02-01 |   31
 2017-02-01 | 2017-02-28 | 2017-03-01 |   28
 2017-03-01 | 2017-03-31 | 2017-04-01 |   31
 2017-04-01 | 2017-04-30 | 2017-05-01 |   30
 2017-05-01 | 2017-05-31 | 2017-06-01 |   31
 2017-06-01 | 2017-06-30 | 2017-07-01 |   30
 2017-07-01 | 2017-07-31 | 2017-08-01 |   31
 2017-08-01 | 2017-08-31 | 2017-09-01 |   31
 2017-09-01 | 2017-09-30 | 2017-10-01 |   30
 2017-10-01 | 2017-10-31 | 2017-11-01 |   31
 2017-11-01 | 2017-11-30 | 2017-12-01 |   30
 2017-12-01 | 2017-12-31 | 2018-01-01 |   31
(12 rows)
```

PostgreSQL's implementation of the calendar is very good, so use it!

Date/Time Processing and Querying

Once the application's data, or rather the user data is properly stored as timestamp with time zone, PostgreSQL allows implementing all the

processing you need to.

As an example data set this time we're playing with *git* history. The PostgreSQL and pgloader project history have been loaded into the *commitlog* table thanks to the *git log* command, with a custom format, and some post-processing — properly splitting up the commit's subjects and escaping its content. Here's for example the most recent commit registered in our local *commitlog* table:

```
  select project, hash, author, ats, committer, cts, subject
    from commitlog
   where project = 'postgresql'
order by ats desc
   limit 1;
```

The column names *ats* and *cts* respectively stand for *author commit timestamp* and *committer commit timestamp*, and the *subject* is the first line of the commit message, as per the *git log* format *%s*.

To get the most recent entry from a table we *order by* dates in *descending* order then *limit* the result set to a single entry, and we get a single line of output:

```
-[ RECORD 1 ]-------------------------------------------------------------
project   | postgresql
hash      | b1c2d76a2fcef812af0be3343082414d401909c8
author    | Tom Lane
ats       | 2017-08-19 19:39:37+02
committer | Tom Lane
cts       | 2017-08-19 19:39:51+02
subject   | Fix possible core dump in parallel restore when using a TOC list.
```

With timestamps, we can compute time-based reporting, such as how many commits each project received each year in their whole history:

```
  select extract(year from ats) as year,
         count(*) filter(where project = 'postgresql') as postgresql,
         count(*) filter(where project = 'pgloader') as pgloader
    from commitlog
group by year
order by year;
```

As we have only loaded two projects in our *commitlog* table, the output is better with a *pivot* query. We can see more than 20 years of sustained activity for the PostgreSQL project, and a less active project for pgloader:

```
 year | postgresql | pgloader
======+============+==========
 1996 |        876 |        0
```

```
 1997 |        1698 |         0
 1998 |        1744 |         0
 1999 |        1788 |         0
 2000 |        2535 |         0
 2001 |        3061 |         0
 2002 |        2654 |         0
 2003 |        2416 |         0
 2004 |        2548 |         0
 2005 |        2418 |         3
 2006 |        2153 |         3
 2007 |        2188 |        42
 2008 |        1651 |        63
 2009 |        1389 |         3
 2010 |        1800 |        29
 2011 |        2030 |         2
 2012 |        1605 |         2
 2013 |        1368 |       385
 2014 |        1745 |       367
 2015 |        1815 |       202
 2016 |        2086 |       136
 2017 |        1721 |       142
(22 rows)
```

We can also build a reporting on the repartition of commits by weekday from the beginning of the project, in order to guess if contributors are working on the project on the job only, or mostly during their free time (weekend).

```
  select extract(isodow from ats) as dow,
         to_char(ats, 'Day') as day,
         count(*) as commits,
         round(100.0*count(*)/sum(count(*)) over(), 2) as pct,
         repeat('■', (100*count(*)/sum(count(*)) over())::int) as hist
    from commitlog
   where project = 'postgresql'
group by dow, day
order by dow;
```

It seems that our PostgreSQL committers tend to work whenever they feel like it, but less so on the weekend. The project's lucky enough to have a solid team of committers being paid to work on PostgreSQL:

```
 dow |    day    | commits |  pct  |       hist
═════╪═══════════╪═════════╪═══════╪══════════════════
   1 | Monday    |    6552 | 15.14 | ■■■■■■■■■■■■■■■
   2 | Tuesday   |    7164 | 16.55 | ■■■■■■■■■■■■■■■■
   3 | Wednesday |    6477 | 14.96 | ■■■■■■■■■■■■■■■
   4 | Thursday  |    7061 | 16.31 | ■■■■■■■■■■■■■■■■
   5 | Friday    |    7008 | 16.19 | ■■■■■■■■■■■■■■■■
   6 | Saturday  |    4690 | 10.83 | ■■■■■■■■■■
```

```
   7 | Sunday    |    4337 | 10.02 | ■■■■■■■■■■
(7 rows)
```

Another report we can build compares the author commit timestamp with the committer commit timestamp. Those are different, but by how much?

```
with perc_arrays as
  (
       select project,
              avg(cts-ats) as average,
              percentile_cont(array[0.5, 0.9, 0.95, 0.99])
                 within group(order by cts-ats) as parr
         from commitlog
        where ats <> cts
     group by project
  )
 select project, average,
        parr[1] as median,
        parr[2] as "%90th",
        parr[3] as "%95th",
        parr[4] as "%99th"
   from perc_arrays;
```

Here's a detailed output of the time difference statistics, per project:

```
-[ RECORD 1 ]-------------------------------------
project | pgloader
average | @ 4 days 22 hours 7 mins 41.18 secs
median  | @ 5 mins 21.5 secs
%90th   | @ 1 day 20 hours 49 mins 49.2 secs
%95th   | @ 25 days 15 hours 53 mins 48.15 secs
%99th   | @ 169 days 24 hours 33 mins 26.18 secs
=[ RECORD 2 ]=====================================
project | postgres
average | @ 1 day 10 hours 15 mins 9.706809 secs
median  | @ 2 mins 4 secs
%90th   | @ 1 hour 46 mins 13.5 secs
%95th   | @ 1 day 17 hours 58 mins 7.5 secs
%99th   | @ 40 days 20 hours 36 mins 43.1 secs
```

Reporting is a strong use case for SQL. Application will also send more classic queries. We can show the commits for the PostgreSQL project for the 1st of June 2017:

```
\set day '2017-06-01'

  select ats::time,
         substring(hash from 1 for 8) as hash,
         substring(subject from 1 for 40) || '···' as subject
    from commitlog
   where project = 'postgresql'
     and ats >= date :'day'
```

```
      and ats  < date :'day' + interval '1 day'
order by ats;
```

It's tempting to use the *between* SQL operator, but we would then have to remember that *between* includes both its lower and upper bound and we would then have to compute the upper bound as the very last instant of the day. Using explicit *greater than or equal* and *less than* operators makes it possible to always compute the very first time of the day, which is easier, and well supported by PostgreSQL.

Also, using explicit bound checks allows ys to use a single date literal in the query, so that's a single parameter to send from the application.

```
   ats     │   hash   │                  subject
═══════════╪══════════╪═══════════════════════════════════════════
 01:39:27  │ 3d79013b │ Make ALTER SEQUENCE, including RESTART, ···
 02:03:10  │ 66510455 │ Modify sequence catalog tuple before inv···
 04:35:33  │ de492c17 │ doc: Add note that DROP SUBSCRIPTION dro···
 19:32:55  │ e9a3c047 │ Always use -fPIC, not -fpic, when buildi···
 23:45:53  │ f112f175 │ Fix typo···
(5 rows)
```

Many data type formatting functions are available in PostgreSQL. In the previous querym although we chose to *cast* our timestamp with time zone entry down to a *time* value, we could have chosen another representation thanks to the *to_char* function:

```
set lc_time to 'fr_FR';

  select to_char(ats, 'TMDay TMDD TMMonth, HHam') as time,
         substring(hash from 1 for 8) as hash,
         substring(subject from 1 for 40) || '···' as subject
    from commitlog
   where project = 'postgresql'
     and ats >= date :'day'
     and ats  < date :'day' + interval '1 day'
order by ats;
```

And this time we have a French localized output for the time value:

```
        time          │   hash   │                  subject
══════════════════════╪══════════╪═══════════════════════════════════════════
 Jeudi 01 Juin, 01am  │ 3d79013b │ Make ALTER SEQUENCE, including RESTART, ···
 Jeudi 01 Juin, 02am  │ 66510455 │ Modify sequence catalog tuple before inv···
 Jeudi 01 Juin, 04am  │ de492c17 │ doc: Add note that DROP SUBSCRIPTION dro···
 Jeudi 01 Juin, 07pm  │ e9a3c047 │ Always use -fPIC, not -fpic, when buildi···
 Jeudi 01 Juin, 11pm  │ f112f175 │ Fix typo···
(5 rows)
```

Take some time to familiarize yourself with the time and date support that PostgreSQL comes with out of the box. Some very useful functions such as *date_trunc()* are not shown here, and you also will find more gems.

While most programming languages nowadays include the same kind of feature set, having this processing feature set right in PostgreSQL makes sense in several use cases:

- It makes sense when the SQL logic or filtering you want to implement depends on the result of the processing (e.g. grouping by week).
- When you have several applications using the same logic, it's often easier to share a SQL query than to set up a distributed service API offering the same result in XML or JSON (a data format you then have to parse).
- When you want to reduce your run-time dependencies, it's a good idea to understand how much each architecture layer is able to support in your implementation.

Network Address Types

PostgreSQL includes support for both *cidr*, *inet*, and *macaddr* data types. Again, those types are bundled with indexing support and advanced functions and operator support.

The PostgreSQL documentation chapters entitled Network Address Types and Network Address Functions and Operators cover network address types.

Web servers logs are a classic source of data to process where we find network address types and The Honeynet Project has some free samples for us to play with. This time we're using the *Scan 34* entry. Here's how to load the sample data set, once cleaned into a proper CSV file:

```
1  begin;
2
3  drop table if exists access_log;
4
5  create table access_log
6   (
7    ip      inet,
8    ts      timestamptz,
```

```
  request text,
  status  integer
 );

\copy access_log from 'access.csv' with csv delimiter ';'

commit;
```

The script used to cleanse the original data into a CSV that PostgreSQL is happy about implements a pretty simple transformation from

```
211.141.115.145 - - [13/Mar/2005:04:10:18 -0500] "GET / HTTP/1.1" 403 2898 "-" "Mozill
```

into

```
"211.141.115.145";"2005-05-13 04:10:18 -0500";"GET / HTTP/1.1";"403"
```

Being mostly interested into network address types, the transformation from the Apache access log format to CSV is lossy here, we keep only some of the fields we might be interested into.

One of the things that's possible to implement thanks to the PostgreSQL *inet* data type is an analysis of */24* networks that are to be found in the logs.

To enable that analysis, we can use the *set_masklen()* function which allows us to transforms an IP address into an arbitrary CIDR network address:

```
select distinct on (ip)
       ip,
       set_masklen(ip, 24) as inet_24,
       set_masklen(ip::cidr, 24) as cidr_24
  from access_log
 limit 10;
```

And we can see that if we keep the data type as *inet*, we still get the full IP address with the */24* network notation added. To have the *.0/24* notation we need to be using *cidr*:

```
      ip       |      inet_24       |     cidr_24
═══════════════╪════════════════════╪═════════════════
 4.35.221.243  | 4.35.221.243/24    | 4.35.221.0/24
 4.152.207.126 | 4.152.207.126/24   | 4.152.207.0/24
 4.152.207.238 | 4.152.207.238/24   | 4.152.207.0/24
 4.249.111.162 | 4.249.111.162/24   | 4.249.111.0/24
 12.1.223.132  | 12.1.223.132/24    | 12.1.223.0/24
 12.8.192.60   | 12.8.192.60/24     | 12.8.192.0/24
 12.33.114.7   | 12.33.114.7/24     | 12.33.114.0/24
 12.47.120.130 | 12.47.120.130/24   | 12.47.120.0/24
```

```
 12.172.137.4  | 12.172.137.4/24  | 12.172.137.0/24
 18.194.1.122  | 18.194.1.122/24  | 18.194.1.0/24
(10 rows)
```

Of course, note that you could be analyzing other networks than */24*:

```
select distinct on (ip)
       ip,
       set_masklen(ip::cidr, 27) as cidr_27,
       set_masklen(ip::cidr, 28) as cidr_28
  from access_log
 limit 10;
```

This computes for us the proper starting ip addresses for our CIDR notation for us, of course. After all, what's the point of using proper data types if not for advanced processing?

```
      ip       |     cidr_27      |     cidr_28
═══════════════╪══════════════════╪══════════════════
 4.35.221.243  | 4.35.221.224/27  | 4.35.221.240/28
 4.152.207.126 | 4.152.207.96/27  | 4.152.207.112/28
 4.152.207.238 | 4.152.207.224/27 | 4.152.207.224/28
 4.249.111.162 | 4.249.111.160/27 | 4.249.111.160/28
 12.1.223.132  | 12.1.223.128/27  | 12.1.223.128/28
 12.8.192.60   | 12.8.192.32/27   | 12.8.192.48/28
 12.33.114.7   | 12.33.114.0/27   | 12.33.114.0/28
 12.47.120.130 | 12.47.120.128/27 | 12.47.120.128/28
 12.172.137.4  | 12.172.137.0/27  | 12.172.137.0/28
 18.194.1.122  | 18.194.1.96/27   | 18.194.1.112/28
(10 rows)
```

Equipped with this *set_masklen()* function, it's now easy to analyze our access logs using arbitrary CIDR network definitions.

```
  select set_masklen(ip::cidr, 24) as network,
         count(*) as requests,
         array_length(array_agg(distinct ip), 1) as ipcount
    from access_log
group by network
  having array_length(array_agg(distinct ip), 1) > 1
order by requests desc, ipcount desc;
```

In our case, we get the following result:

```
    network      | requests | ipcount
═════════════════╪══════════╪═════════
 4.152.207.0/24  |      140 |       2
 222.95.35.0/24  |       59 |       2
 211.59.0.0/24   |       32 |       2
 61.10.7.0/24    |       25 |      25
 222.166.160.0/24|       25 |      24
```

```
 219.153.10.0/24  |       7 |       3
 218.78.209.0/24  |       6 |       4
 193.109.122.0/24 |       5 |       5
 204.102.106.0/24 |       3 |       3
 66.134.74.0/24   |       2 |       2
 219.133.137.0/24 |       2 |       2
 61.180.25.0/24   |       2 |       2
(12 rows)
```

Ranges

Range types are a unique feature of PostgreSQL, managing two dimensions of data in a single column, and allowing advanced processing. The main example is the *daterange* data type, which stores as a single value a lower and an upper bound of the range as a single value. This allows PostgreSQL to implement a concurrent safe check against *overlapping* ranges, as we're going to see in the next example.

As usual, read the PostgreSQL documentation chapters with the titles Range Types and Range Functions and Operators for complete information.

The International Monetary Fund publishes exchange rate archives by month for lots of currencies. An exchange rate is relevant from its publication until the next rate is published, which makes a very good use case for our PostgreSQL range types.

The following SQL script is the main part of the *ELT* script that has been used for this book. Only missing from this book's pages is the transformation script that pivots the available *tsv* file into the more interesting format we use here:

```
1  begin;
2
3  create schema if not exists raw;
4
5  -- Must be run as a Super User in your database instance
6  -- create extension if not exists btree_gist;
7
8  drop table if exists raw.rates, rates;
9
10 create table raw.rates
11  (
12   currency text,
13   date     date,
```

```
  rate     numeric
 );

\copy raw.rates from 'rates.csv' with csv delimiter ';'

create table rates
 (
  currency text,
  validity daterange,
  rate     numeric,

  exclude using gist (currency with =,
                      validity with &&)
 );

insert into rates(currency, validity, rate)
     select currency,
            daterange(date,
                      lead(date) over(partition by currency
                                          order by date),
                      '[)'
                     )
            as validity,
            rate
       from raw.rates
   order by date;

commit;
```

In this SQL script, we first create a target table for loading the CSV file. The file contains lines with a currency name, a date of publication, and a rate as a *numeric* value. Once the data is loaded into this table, we can transform it into something more interesting to work with from an application, the *rates* table.

The *rates* table registers the rate value for a currency and a *validity* period, and uses an exclusion constraint that guarantees non-overlapping *validity* periods for any given *currency*:

```
exclude using gist (currency with =, validity with &&)
```

This expression reads: exclude any tuple where the currency is = to an existing currency in our table *AND* where the *validity* is overlapping with (&&) any existing validity in our table. This exclusion constraint is implemented in PostgreSQL using a *GiST* index.

By default, *GiST* in PostgreSQL doesn't support one-dimensional data types that are meant to be covered by *B-tree* indexes. With exclusion

constraints though, it's very interesting to extend *GiST* support for one-dimensional data types, and so we install the *btree_gist* extension, provided in PostgreSQL contrib package.

The script then fills in the *rates* table from the *raw.rates* we'd been importing in the previous step. The query uses the *lead()* window function to implement the specification spelled out in English earlier: *an exchange rate is relevant from its publication until the next rate is published.*

Here's how the data looks, with the following query targeting Euro rates:

```
  select currency, validity, rate
    from rates
   where currency = 'Euro'
order by validity
   limit 10;
```

We can see that the validity is a range of dates, and the standard output for this type is a closed range which includes the first entry and excludes the second one:

```
 currency │        validity         │   rate
══════════╪═════════════════════════╪══════════
 Euro     │ [2017-05-02,2017-05-03) │ 1.254600
 Euro     │ [2017-05-03,2017-05-04) │ 1.254030
 Euro     │ [2017-05-04,2017-05-05) │ 1.252780
 Euro     │ [2017-05-05,2017-05-08) │ 1.250510
 Euro     │ [2017-05-08,2017-05-09) │ 1.252880
 Euro     │ [2017-05-09,2017-05-10) │ 1.255280
 Euro     │ [2017-05-10,2017-05-11) │ 1.255300
 Euro     │ [2017-05-11,2017-05-12) │ 1.257320
 Euro     │ [2017-05-12,2017-05-15) │ 1.255530
 Euro     │ [2017-05-15,2017-05-16) │ 1.248960
(10 rows)
```

Having this data set with the exclusion constraint means that we know we have at most a single rate available at any point in time, which allows an application needing the rate for a specific time to write the following query:

```
select rate
  from rates
 where currency = 'Euro'
   and validity @> date '2017-05-18';
```

The operator @> reads *contains*, and PostgreSQL uses the exclusion constraint's index to solve that query efficiently:

```
  rate
```

```
 1.240740
(1 row)
```

Denormalized Data Types

The main idea behind the PostgreSQL project from Michael Stonebraker has been *extensibility*. As a result of that design choice, some data types supported by PostgreSQL allow bypassing relational constraint. For instance, PostgreSQL supports *arrays*, which store several values in the same attribute value. In standard SQL, the content of the *array* would be completely opaque, so the array would be considered only as a whole.

The extensible design of PostgreSQL makes it possible to enrich the SQL language with new capabilities. Specific operators are built for denormalized data types and allow addressing values contained into an *array* or a *json* attribute value, integrating perfectly with SQL.

The following data types are built-in to PostgreSQL and extend its processing capabilities to another level.

Arrays

PostgreSQL has built-in support for arrays, which are documented in the Arrays and the Array Functions and Operators chapters. As introduced above, what's interesting with PostgreSQL is its ability to process array elements from SQL directly. This capability includes indexing facilities thanks to GIN indexing.

Arrays can be used to denormalize data and avoid lookup tables. A good rule of thumb for using them that way is that you mostly use the array as a whole, even if you might at times search for elements in the array. Heavier processing is going to be more complex than a lookup table.

A classic example of a good use case for PostgreSQL arrays is user-defined tags. For the next example, 200,000 USA geolocated tweets have been loaded into PostgreSQL thanks to the following script:

```
begin;

create table tweet
 (
   id          bigint primary key,
   date        date,
   hour        time,
   uname       text,
   nickname    text,
   bio         text,
   message     text,
   favs        bigint,
   rts         bigint,
   latitude    double precision,
   longitude   double precision,
   country     text,
   place       text,
   picture     text,
   followers   bigint,
   following   bigint,
   listed      bigint,
   lang        text,
   url         text
 );

\copy tweet from 'tweets.csv' with csv header delimiter ';'

commit;
```

Once the data is loaded we can have a look at it:

```
\pset format wrapped
\pset columns 70
table tweet limit 1;
```

Here's what it looks like:

```
-[ RECORD 1 ]-------------------------------------------------------
id         | 721318437075685382
date       | 2016-04-16
hour       | 12:44:00
uname      | Bill Schulhoff
nickname   | BillSchulhoff
bio        | Husband,Dad,GrandDad,Ordained Minister, Umpire, Poker Pla···
           |···yer, Mets, Jets, Rangers, LI Ducks, Sons of Anarchy, Surv···
           |···ivor, Apprentice, O&A, & a good cigar
message    | Wind 3.2 mph NNE. Barometer 30.20 in, Rising slowly. Temp···
           |···erature 49.3 °F. Rain today 0.00 in. Humidity 32%
favs       | ¤
rts        | ¤
latitude   | 40.76027778
longitude  | -72.95472222
```

```
country   | US
place     | East Patchogue, NY
picture   | http://pbs.twimg.com/profile_images/378800000718469152/53···
          |···5032cf772ca04524e0fe075d3b4767_normal.jpeg
followers | 386
following | 705
listed    | 24
lang      | en
url       | http://www.twitter.com/BillSchulhoff/status/7213184370756···
          |···85382
```

We can see that the raw import schema is not a good fit for PostgreSQL capabilities. The *date* and *hour* fields are separated for no good reason, and it makes processing them less easy than when they form a *timestamptz* together. PostgreSQL does know how to handle *longitude* and *latitude* as a single *point* entry, allowing much more interesting processing again. We can create a simpler relation to manage and process a subset of the data we're interested in for this chapter.

As we are interested in the tags used in the messages, the next query also extracts all the tags from the Twitter messages as an array of text.

```
1  begin;
2
3  create table hashtag
4   (
5     id          bigint primary key,
6     date        timestamptz,
7     uname       text,
8     message     text,
9     location    point,
10    hashtags    text[]
11  );
12
13 with matches as (
14   select id,
15          regexp_matches(message, '(#[^ ,]+)', 'g') as match
16     from tweet
17 ),
18     hashtags as (
19   select id,
20          array_agg(match[1] order by match[1]) as hashtags
21     from matches
22 group by id
23 )
24 insert into hashtag(id, date, uname, message, location, hashtags)
25      select id,
26             date + hour as date,
27             uname,
```

```
28 |            message,
29 |            point(longitude, latitude),
30 |            hashtags
31 |       from      hashtags
32 |            join tweet using(id);
33 |
34 | commit;
```

The PostgreSQL matching function *regexp_matches()* implements what we need here, with the *g* flag to return every match found and not just the first tag in a message. Those multiple matches are returned one per row, so we then *group by* tweet id and *array_agg* over them, building our array of tags. Here's what the computed data looks like:

```
1 | select id, hashtags
2 |   from hashtag
3 |  limit 10;
```

In the following data output, you can see that we kept the # signs in front of the hashtags, making it easier to recognize what this data is:

```
         id          |                    hashtags
═════════════════════╪═══════════════════════════════════════════════
 720553447402160128  | {#CriminalMischief,#ocso,#orlpol}
 720553457015324672  | {#txwx}
 720553458596757504  | {#DrugViolation,#opd,#orlpol}
 720553466804989952  | {#Philadelphia,#quiz}
 720553475923271680  | {#Retail,#hiring!,#job}
 720553508190052352  | {#downtown,#early⋯,#ghosttown,#longisland,#morn⋯
                     |⋯ing,#portjeff,#portjefferson}
 720553522966581248  | {"#CapitolHeights,",#Retail,#hiring!,#job}
 720553530088669185  | {#NY17}
 720553531665682434  | {#Endomondo,#endorphins}
 720553532273795072  | {#Job,#Nursing,"#Omaha,",#hiring!}
(10 rows)
```

Before processing the tags, we create a specialized *GIN* index. This index access method allows PostgreSQL to index the *contents* of the arrays, the tags themselves, rather than each array as an opaque value.

```
1 | create index on hashtag using gin (hashtags);
```

A popular tag in the dataset is *#job*, and we can easily see how many times it's been used, and confirm that our previous index makes sense for looking inside the *hashtags* array:

```
1 | explain (analyze, verbose, costs off, buffers)
2 |  select count(*)
3 |   from hashtag
4 |  where hashtags @> array['#job'];
```

```
                                QUERY PLAN
═══════════════════════════════════════════════════════════════════════════
 Aggregate (actual time=27.227..27.227 rows=1 loops=1)
   Output: count(*)
   Buffers: shared hit=3715
   ->  Bitmap Heap Scan on public.hashtag (actual time=13.023..23.453⋯
⋯ rows=17763 loops=1)
         Output: id, date, uname, message, location, hashtags
         Recheck Cond: (hashtag.hashtags @> '{#job}'::text[])
         Heap Blocks: exact=3707
         Buffers: shared hit=3715
         ->  Bitmap Index Scan on hashtag_hashtags_idx (actual time=1⋯
⋯1.030..11.030 rows=17763 loops=1)
               Index Cond: (hashtag.hashtags @> '{#job}'::text[])
               Buffers: shared hit=8
 Planning time: 0.596 ms
 Execution time: 27.313 ms
(13 rows)
```

That was done supposing we already know one of the popular tags. How do we get to discover that information, given our data model and data set? We do it with the following query:

```
  select tag, count(*)
    from hashtag, unnest(hashtags) as t(tag)
group by tag
order by count desc
   limit 10;
```

This time, as the query must scan all the hashtags in the table, it won't use the previous index of course. The *unnest()* function is a must-have when dealing with arrays in PostgreSQL, as it allows processing the array's content as if it were just another relation. And SQL comes with all the tooling to process relations, as we've already seen in this book.

So we can see the most popular hashtags in our dataset:

```
     tag      │ count
══════════════╪═══════
 #Hiring      │ 37964
 #Jobs        │ 24776
 #CareerArc   │ 21845
 #Job         │ 21368
 #job         │ 17763
 #Retail      │  7867
 #Hospitality │  7664
 #job?        │  7569
 #hiring!     │  6860
 #Job:        │  5953
(10 rows)
```

The hiring theme is huge in this dataset. We could then search for mentions of job opportunities in the *#Retail* sector (another popular hashtag we just discovered into the data set), and have a look at the locations where they are saying they're hiring:

```
  select name,
         substring(timezone, '/(.*)') as tz,
         count(*)
    from hashtag

         left join lateral
         (
            select *
              from geonames
          order by location <-> hashtag.location
             limit 1
         )
         as geoname
         on true

   where hashtags @> array['#Hiring', '#Retail']

group by name, tz
order by count desc
   limit 10;
```

For this query a dataset of *geonames* has been imported. The *left join lateral* allows picking the nearest location to the tweet location from our *geoname* reference table. The *where* clause only matches the hashtag arrays containing both the *#Hiring* and the *#Retail* tags. Finally, we order the data set by most promising opportunities:

```
                    name                    |     tz      | count
============================================+=============+=======
 San Jose City Hall                         | Los_Angeles |    31
 Sleep Inn & Suites Intercontinental Airport East | Chicago |    19
 Los Angeles                                | Los_Angeles |    14
 Dallas City Hall Plaza                     | Chicago     |    12
 New York City Hall                         | New_York    |    11
 Jw Marriott Miami Downtown                 | New_York    |    11
 Gold Spike Hotel & Casino                  | Los_Angeles |    10
 San Antonio                                | Chicago     |    10
 Shoppes at 104                             | New_York    |     9
 Fruitville Elementary School               | New_York    |     8
(10 rows)
```

PostgreSQL arrays are very powerful, and GIN indexing support makes them efficient to work with. Nonetheless, it's still not so efficient that you would replace a lookup table with an array in situations where you do a

lot of lookups, though.

Also, some PostgreSQL array functions show a quadratic behavior: looping over arrays elements really is inefficient, so learn to use *unnest()* instead, and filter elements with a *where* clause. If you see yourself doing that a lot, it might be a good sign that you really needed a lookup table!

Composite Types

PostgreSQL tables are made of tuples with a known type. It is possible to manage that type separately from the main table, as in the following script:

```
1  begin;
2
3  create type rate_t as
4   (
5     currency text,
6     validity daterange,
7     value    numeric
8   );
9
10 create table rate of rate_t
11  (
12    exclude using gist (currency with =,
13                        validity with &&)
14  );
15
16 insert into rate(currency, validity, value)
17      select currency, validity, rate
18        from rates;
19
20 commit;
```

The *rate* table works exactly like the *rates* one that we defined earlier in this chapter.

```
1  table rate limit 10;
```

We get the kind of result we expect:

```
      currency       |        validity         |    value
═════════════════════╪═════════════════════════╪═══════════
 New Zealand Dollar  | [2017-05-01,2017-05-02) |    1.997140
 Colombian Peso      | [2017-05-01,2017-05-02) | 4036.910000
 Japanese Yen        | [2017-05-01,2017-05-02) |  152.624000
 Saudi Arabian Riyal | [2017-05-01,2017-05-02) |    5.135420
 Qatar Riyal         | [2017-05-01,2017-05-02) |    4.984770
```

```
 Chilean Peso        | [2017-05-01,2017-05-02) |   911.245000
 Rial Omani          | [2017-05-01,2017-05-02) |     0.526551
 Iranian Rial        | [2017-05-01,2017-05-02) | 44426.100000
 Bahrain Dinar       | [2017-05-01,2017-05-02) |     0.514909
 Kuwaiti Dinar       | [2017-05-01,2017-05-02) |     0.416722
(10 rows)
```

It is interesting to build composite types in advanced cases, which are not covered in this book, such as:

- Management of *Stored Procedures* API
- Advanced use cases of *array* of *composite* types

XML

The SQL standard includes a SQL/XML which *introduces the predefined data type XML together with constructors, several routines, functions, and XML-to-SQL data type mappings to support manipulation and storage of XML in a SQL database*, as per the Wikipedia page.

PostgreSQL implements the XML data type, which is documented in the chapters on XML type and XML functions chapters.

The best option when you need to process XML documents might be the XSLT transformation language for XML. It should be no surprise that a PostgreSQL extension allows writing *stored procedures* in this language. If you have to deal with XML documents in your database, check out PL/XSLT.

An example of a *PL/XSLT* function follows:

```
1  create extension plxslt;
2
3  CREATE OR REPLACE FUNCTION striptags(xml) RETURNS text
4      LANGUAGE xslt
5  AS $$<?xml version="1.0"?>
6  <xsl:stylesheet version="1.0"
7      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
8      xmlns="http://www.w3.org/1999/xhtml"
9  >
10
11   <xsl:output method="text" omit-xml-declaration="yes"/>
12
13   <xsl:template match="/">
14     <xsl:apply-templates/>
15   </xsl:template>
```

```
16
17 </xsl:stylesheet>
18 $$;
```

It can be used like this:

```
1  create table docs
2   (
3     id      serial primary key,
4     content xml
5   );
6
7  insert into docs(content)
8       values ('<?xml version="1.0"?>
9  <html xmlns="http://www.w3.org/1999/xhtml">
10 <body>hello</body>
11 </html>');
12
13 select id, striptags(content)
14   from docs;
```

As expected, here's the result:

```
 id │ striptags
════╪═══════════
  1 │           ↵
    │ hello     ↵
    │
(1 row)
```

The XML support in PostgreSQL might be handy in cases. It's mainly been added for standard compliance, though, and is not found a lot in the field. XML processing function and XML indexing is pretty limited in PostgreSQL.

JSON

PostgreSQL has built-in support for JSON with a great range of processing functions and operators, and complete indexing support. The documentation covers all the details in the chapters entitled JSON Types and JSON Functions and Operators.

PostgreSQL implemented a very simple *JSON* datatype back in the 9.2 release. At that time the community pushed for providing a solution for *JSON* users, in contrast to the usual careful pace, though still speedy. The

JSON datatype is actually *text* under the hood, with a verification that the format is valid *json* input... much like *XML*.

Later, the community realized that the amount of *JSON* processing and advanced searching required in PostgreSQL would not be easy or reasonable to implement over a text datatype, and implemented a *binary* version of the *JSON* datatype, this time with a full set of operators and functions to work with.

There are some incompatibilities in between the text-based *json* datatype and the newer *jsonb* version of it, where it's been argued that *b* stands for *better*:

- The *json* datatype, being a text datatype, stores the data presentation exactly as it is sent to PostgreSQL, including whitespace and indentation, and also multiple-keys when present (no processing at all is done on the content, only form validation).
- The *jsonb* datatype is an advanced binary storage format with full processing, indexing and searching capabilities, and as such preprocesses the JSON data to an internal format, which does include a single value per key; and also isn't sensible to extra whitespace or indentation.

The data type you probably need and want to use is *jsonb*, not the *json* early draft that is still available for backward compatibility reasons only. Here's a very quick example showing some differences between those two datatypes:

```
create table js(id serial primary key, extra json);
insert into js(extra)
     values ('[1, 2, 3, 4]'),
            ('[2, 3, 5, 8]'),
            ('{"key": "value"}');
```

The *js* table only has a primary key and a *json* column for extra information. It's not a good design, but we want a very simple example here and won't be coding any application on top of it, so it will do for the following couple SQL queries:

```
select * from js where extra @> '2';
```

When we want to search for entries where the *extra* column contains a number in its array, we get the following error:

```
ERROR:  operator does not exist: json @> unknown
LINE 1: select * from js where extra @> '2';
                                     ^
HINT:  No operator matches the given name and argument type(s). ⏎
You might need to add explicit type casts.
```

Right. *json* is only text and not very powerful, and it doesn't offer an implementation for the *contains* operator. Switching the content to *jsonb* then:

```
1 alter table js alter column extra type jsonb;
```

Now we can run the same query again:

```
1 select * from js where extra @> '2';
```

And we find out that of course our sample data set of two rows contains the number *2* in the extra *jsonb* field, which here only contains arrays of numbers:

```
 id │    extra
════╪═════════════
  1 │ [1, 2, 3, 4]
  2 │ [2, 3, 5, 8]
(2 rows)
```

We can also search for JSON arrays containing another JSON array:

```
1 select * from js where extra @> '[2,4]';
```

This time a single row is found, as expected:

```
 id │    extra
════╪═════════════
  1 │ [1, 2, 3, 4]
(1 row)
```

Two use cases for JSON in PostgreSQL are very commonly found:

- The application needs to manage a set of documents that happen to be formatted in *JSON*.
- Application designers and developers aren't too sure about the exact set of fields needed for a part of the data model, and want this data model to be very easily extensible.

In the first case, using *jsonb* is a great enabler in terms of your application's capabilities to process the documents it manages, including searching and filtering using the content of the document. See jsonb Indexing in the PostgreSQL documentation for more information about the `jsonb_path_ops`

which can be used as in the following example and provides a very good general purpose index for the `@>` operator as used in the previous query:

```
create index on js using gin (extra jsonb_path_ops);
```

Now, it is possible to use *jsonb* as a flexible way to maintain your data model. It is possible to then think of PostgreSQL like a *schemaless* service and have a heterogeneous set of documents all in a single relation.

This trade-off sounds interesting from a model design and maintenance perspective, but is very costly when it comes to daily queries and application development: you never really know what you're going to find out in the *jsonb* columns, so you need to be very careful about your SQL statements as you might easily miss rows you wanted to target, for example.

A good trade-off is to design a model with some static columns are created and managed traditionally, and an *extra* column of *jsonb* type is added for those things you didn't know yet, and that would be used only sometimes, maybe for debugging reasons or special cases.

This works well until the application's code is querying the *extra* column in every situation because some important data is found only there. At this point, it's worth promoting parts of the *extra* field content into proper PostgreSQL attributes in your relational schema.

Enum

This data type has been added to PostgreSQL in order to make it easier to support migrations from MySQL. Proper relational design would use a reference table and a foreign key instead:

```
create table color(id serial primary key, name text);

create table cars
 (
   brand   text,
   model   text,
   color   integer references color(id)
 );

insert into color(name)
     values ('blue'), ('red'),
            ('gray'), ('black');

```

```
insert into cars(brand, model, color)
     select brand, model, color.id
      from (
            values('ferari', 'testarosa', 'red'),
                  ('aston martin', 'db2', 'blue'),
                  ('bentley', 'mulsanne', 'gray'),
                  ('ford', 'T', 'black')
           )
             as data(brand, model, color)
           join color on color.name = data.color;
```

In this setup the table *color* lists available colors to choose from, and the cars table registers availability of a model from a brand in a given color. It's possible to make an *enum* type instead:

```
create type color_t as enum('blue', 'red', 'gray', 'black');

drop table if exists cars;
create table cars
 (
   brand   text,
   model   text,
   color   color_t
 );

insert into cars(brand, model, color)
     values ('ferari', 'testarosa', 'red'),
            ('aston martin', 'db2', 'blue'),
            ('bentley', 'mulsanne', 'gray'),
            ('ford', 'T', 'black');
```

Be aware that in MySQL there's no *create type* statement for *enum* types, so each column using an *enum* is assigned its own data type. As you now have a separate anonymous data type per column, good luck maintaining a globally consistent state if you need it.

Using the *enum* PostgreSQL facility is mostly a matter of taste. After all, join operations against small reference tables are well supported by the PostgreSQL SQL engine.

PostgreSQL Extensions

The PostgreSQL contrib modules are a collection of additional features for your favorite RDBMS. In particular, you will find there extra data types such as *hstore*, *ltree*, *earthdistance*, *intarray* or *trigrams*. You should

definitely check out the *contribs* out and have them available in your production environment.

Some of the *extensions* provided in the contrib sections are production diagnostic tools, and you will be happy to have them on hand the day you need them, without having to convince your production engineering team that they can trust the package: they can, and it's easier for them to include it from the get-go. Make it so that *postgresql-contribs* is deployed for your development environments.

PostgreSQL extensions are nonetheless not covered in this book.

An interview with Grégoire Hubert

Grégoire Hubert has been a web developer for about as long as we have had web applications, and his favorite web tooling is found in the PHP ecosystem. He wrote POMM to help better integrate PostgreSQL and PHP better. POMM provides developers with unlimited access to SQL and database features while proposing a high-level API over low-level drivers.

Figure 5.1: The Postgresql object model manager for PHP

Considering that you have different layers of code in a web application, for example client-side JavaScript, backend-side PHP and SQL, what do you think should be the role of each layer?

> *Web applications are historically built on a pile of layers that can be seen as an information chain. At one end there is the*

client that can run a local application in JavaScript, at the other end, there is the database. The client calls an application server either synchronously or asynchronously through an HTTP web service most of the time. This data exchange is interesting because data are highly denormalized and shaped to fit business needs in the browser. The application server has the tricky job to store the data and shape them as needed by the client. There are several patterns to do that, the most common is the Model/View/Controller also known as MVC. In this architecture, the task of dealing with the database is handed to the model layer.

In terms of business logic, having a full-blown programming language both on the client side and on the server-side makes it complex to decide where to implement what, at times. And there's also this SQL programming language on the database side. How much of your business logic would you typically hand off to PostgreSQL?

I am essentially dealing with SQL & PHP on a server side. PHP is an object-oriented imperative programming language which means it is good at execution control logic. SQL is a set-oriented declarative programming language and is perfect for data computing. Knowing this, it is easily understandable that business workflow and data shaping must be made each in its layer. The tricky question is not which part of the business logic should be handled by what but how to mix efficiently these two paradigms (the famous impedance mismatch known to ORM users) and this is what the Pomm Model Manager is good at. Separating business control from data computation also explains why I am reluctant to use database vendor procedural languages.

At the database layer we have to consider both the data model and the queries. How do you deal with relational constraints? What are the benefits and drawbacks of those constraints when compared to a "schemaless" approach?

The normal form guaranties consistency over time. This is critical for business-oriented applications. Surprisingly, only a few people know how to use the normal form, most of the

time, it ends up in a bunch of tables with one primary key per relation. It is like tables were spreadsheets because people focus on values. Relational databases are by far more powerful than that as they emphasize types. Tables are type definitions. With that approach in mind, interactions between tuples can easily be addressed. All types life cycles can be modeled this way. Modern relational databases offer a lot of tools to achieve that, the most powerful being ACID transactions.

Somehow, for a long time, the normal form was a pain when it was to represent extensible data. Most of the time, this data had no computation on them but they still had to be searchable and at least ... here. The support of unstructured types like XML or JSON in relational databases is a huge step forward in focusing on what's really important. Now, in one field there can be labels with translation, multiple postal addresses, media definitions, etc. that were creating a lot of noise in the database schemas before. These are application-oriented structures. It means the database does not have to care about their consistency and they are complex business structure for the application layer.

Integrating SQL in your application's source code can be quite tricky. How do you typically approach that?

It all started from here. Pomm's approach was about finding a way to mix SQL & PHP in order to leverage Postgres features in applications. Marrying application object oriented with relational is not easy, the most significant step is to understand that since SQL uses a projection (the list of fields in a SELECT) to transform the returned type, entities had to be flexible objects. They had to be database ignorants. This is the complete opposite of the Active Record design pattern. Since it is not possible to perform SQL queries from entities it becomes difficult to have nested loops. The philosophy is really simple: call the method that performs the most efficient query for your needs, it will return an iterator on results that will pop flexible (yet typed) entities. Each entity has one or more model classes that define custom queries and a default projection shared by theses queries. Furthermore, it is very convenient to write SQL

queries and use a placeholder in place of the list of fields of the main SELECT.

6 Data Modeling

Contents

As a developer using PostgreSQL one of the most important tasks you have to deal with is modeling the database schema for your application. In order to achieve a solid design, it's important to understand how the schema is then going to be used as well as the trade-offs it involves.

> *Show me your flowcharts and conceal your tables, and I shall continue to be mystified. Show me your tables, and I won't usually need your flowcharts; they'll be obvious.*
>
> ***Fred Brooks***

Depending on the schema you choose for your application, some business cases are going to be easier to solve than others, and given the wrong set

of trade-offs, some SQL queries turn out to be really difficult to write... or impossible to achieve in a single query with an acceptable level of performances.

As with application code design, the database model should be meant for the normal business it serves. As Alan Kay put it *simple things should be simple, complex things should be possible*. You know your database schema is good when all the very simple business cases turn out to be implemented as rather simple SQL queries, yet it's still possible to address very specific advanced needs in reporting or fraud detection, or accounting oddities.

In this book, the data modeling chapter comes quite late for this reason: the testing of a database model is done by writing SQL queries for it, with real-world application and business use cases to answer at the *psql* prompt. Now that we've seen what can be done in SQL with basic, standard and advanced features of PostgreSQL, it makes sense to dive into database modeling.

Object Relational Mapping

Designing a database model reminds one of designing an application's object model, to some degree. This is so much the case that sometimes you might wonder if maintaining both is a case of violating the Don't Repeat Yourself (or *DRY*) principle.

There's a fundamental difference between the application's design of its internal state (object-oriented or not) and the database model, though:

- The application implements workflows, user stories, ways to interact with the system with presentation layers, input systems, event collection APIs and other dynamic and user-oriented activities.
- The database model ensures a consistent view of the whole world at all times, allowing every actor to mind their own business and protecting them from each other so that the world you are working with continues to make sense as a whole.

As a consequence, the object model of the application is best when it's specific to a set of *user stories* making up a solid part of the whole product.

For example, in a marketplace application, the user publication system is dedicated to getting information from the user and making it available to other users. The object model for this part of the application might need pricing information, but it knows nothing about the customer's invoicing system.

The database model must ensure that every user action being paid for is accounted for correctly, and invoiced appropriately to the right party, either via internal booking or sent to customers. Invoicing usually implements rules for VAT by country, depending on the kind of goods as well as if the buyer is a company or an individual.

Maintaining a single *object model* for the whole application tends to lead to monolith application design and to reduced modularity, which then slows down the development and accelerates technical debt.

Best practice application design separates user workflow from systemic consistency, and *transactions* have been invented as a mechanism to implement the latter. Your *relational database management system* is meant to be part of your application design, ensuring a consistent world at all times.

Database modeling is very different from object modeling. There are reliable snapshots of a constantly evolving world on the one side, and transient in-flights workflows on the other side.

Tooling for Database Modeling

The psql tool implements the SQL *REPL* for PostgreSQL and supports the whole set of SQL languages, including *data definition language*. It's then possible to have immediate feedback on some design choices or to check out possibilities and behaviors right from the console.

Visual display of a database model tends to be helpful too, in particular to understand the model when first exposed to it.

The database schema is living with your application and business and as such it needs versioning and maintenance. New tables are going to be implemented to support new products, and existing relations are going to evolve in order to support new product features, too.

As with code that is deployed and used, adding features while retaining compatibility to existing use cases is much harder and time consuming than writing the first version. And the first version usually is an MVP of sorts, much simpler than the Real Thing anyway.

To cater to needs associated with long-term maintenance we need versioning. Here, it is schema versioning in production, and also versioning of the *source code* of your database schema. Naturally, this is easily achieved when using SQL files to handle your schema, of course.

Some visual tools allow one to connect to an existing database schema and prepare the visual documentation from the tables and constraints (*primary keys*, *foreign keys*, etc) found in the PostgreSQL catalogs. Those tools allow for both production ready schema versioning and visual documentation.

In this book, we focus on the schema itself rather than its visual representation, so this chapter contains SQL code that you can version control together with your application's code.

How to Write a Database Model

In the writing SQL queries chapter we saw how to write SQL queries as separate `.sql` files, and we learnt about using query parameters with the *psql* syntax for that (`:variable`, `:'variable'`, and `:"identifier"`). For writing our database model, the same tooling is all we need. An important aspect of using *psql* is its capacity to provide immediate feedback, and we can also have that with modeling too.

```
create database sandbox;
```

Now you have a place where to try things out without disturbing existing application code. If you need to interact with existing SQL objects, it might be better to use a *schema* rather than a full-blown separate database:

```
create schema sandbox;
set search_path to sandbox;
```

In PostgreSQL, each database is an isolated environment. A connection string must pick a target database, and it's not possible for one database to interact with objects from another one, because catalogs are kept separated. This is great for isolation purposes. If you want to be able to *join*

data in between your *sandbox* and your application models, use a *schema* instead.

When trying a new schema, it's nice to be able to refine it as you go, trying things out. Here's a simple and effective trick to enable that: write your schema as a SQL script with explicit transaction control, and finish it with your testing queries and a *rollback*.

In the following example, we iterate over the definition of a schema for a kind of forum application about the news. Articles are written and tagged with a single category, which is selected from a curated list that is maintained by the editors. Users can read the articles, of course, and comment on them. In this *MVP*, it's not possible to comment on a comment.

We would like to have a schema and a data set to play with, with some categories, an interesting number of articles and a random number of comments for each article.

Here's a SQL script that creates the first version of our schema and populates it with random data following the specifications above, which are intentionally pretty loose. Notice how the script is contained within a single transaction and ends with a *rollback* statement: PostgreSQL even implements transaction for DDL statements.

```
1  begin;
2
3  create schema if not exists sandbox;
4
5  create table sandbox.category
6   (
7     id    serial primary key,
8     name  text not null
9   );
10
11 insert into sandbox.category(name)
12      values ('sport'),('news'),('box office'),('music');
13
14 create table sandbox.article
15  (
16    id          bigserial primary key,
17    category    integer references sandbox.category(id),
18    title       text not null,
19    content     text
20  );
21
22 create table sandbox.comment
23  (
```

```
   id         bigserial primary key,
   article    integer references sandbox.article(id),
   content    text
 );

insert into sandbox.article(category, title, content)
     select random(1, 4) as category,
            initcap(sandbox.lorem(5)) as title,
            sandbox.lorem(100) as content
       from generate_series(1, 1000) as t(x);

insert into sandbox.comment(article, content)
     select random(1, 1000) as article,
            sandbox.lorem(150) as content
       from generate_series(1, 50000) as t(x);

select article.id, category.name, title
  from      sandbox.article
       join sandbox.category
         on category.id = article.category
 limit 3;

select count(*),
       avg(length(title))::int as avg_title_length,
       avg(length(content))::int as avg_content_length
  from sandbox.article;

   select article.id, article.title, count(*)
     from      sandbox.article
          join sandbox.comment
            on article.id = comment.article
group by article.id
order by count desc
   limit 5;

select category.name,
       count(distinct article.id) as articles,
       count(*) as comments
  from      sandbox.category
       left join sandbox.article on article.category = category.id
       left join sandbox.comment on comment.article = article.id
group by category.name
order by category.name;

rollback;
```

This SQL script references ad-hoc functions creating a random data set. This time for the book I've been using a source of *Lorem Ipsum* texts and some variations on the *random()* function. Typical usage of the script would be at the *psql* prompt thanks to the `\i` command:

```
yesql# \i .../path/to/schema.sql
BEGIN
...
CREATE TABLE
INSERT 0 4
CREATE TABLE
CREATE TABLE
INSERT 0 1000
INSERT 0 50000
 id |    name     |                   title
════╪═════════════╪═══════════════════════════════════════════
  1 | sport       | Debitis Sed Aperiam Id Ea
  2 | sport       | Aspernatur Elit Cumque Sapiente Eiusmod
  3 | box office | Tempor Accusamus Quo Molestiae Adipisci
(3 rows)

 count | avg_title_length | avg_content_length
═══════╪══════════════════╪════════════════════
  1000 |               35 |                738
(1 row)

 id  |                     title                      | count
═════╪════════════════════════════════════════════════╪═══════
 187 | Quos Quaerat Ducimus Pariatur Consequatur      |    73
 494 | Inventore Eligendi Natus Iusto Suscipit        |    73
 746 | Harum Saepe Hic Tempor Alias                   |    70
 223 | Fugiat Sed Dolorum Expedita Sapiente           |    69
 353 | Dignissimos Tenetur Magnam Quaerat Suscipit    |    69
(5 rows)

    name     | articles | comments
═════════════╪══════════╪══════════
 box office |      322 |    16113
 music       |      169 |     8370
 news        |      340 |    17049
 sport       |      169 |     8468
(4 rows)

ROLLBACK
```

As the script ends with a *ROLLBACK* command, you can now edit your schema and do it again, at will, without having to first clean up the previous run.

Generating Random Data

In the previous script, you might have noticed calls to functions that don't exist in the distribution of PostgreSQL, such as *random(int, int)* or *sandbox.lorem(int)*. Here's a complete ad-hoc definition for them:

```
begin;

create schema if not exists sandbox;

drop table if exists sandbox.lorem;

create table sandbox.lorem
 (
   word text
 );

with w(word) as
(
     select regexp_split_to_table('Lorem ipsum dolor sit amet, consectetur
        adipiscing elit, sed do eiusmod tempor incididunt ut labore et
        dolore magna aliqua. Ut enim ad minim veniam, quis nostrud
        exercitation ullamco laboris nisi ut aliquip ex ea commodo
        consequat. Duis aute irure dolor in reprehenderit in voluptate velit
        esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat
        cupidatat non proident, sunt in culpa qui officia deserunt mollit
        anim id est laborum.'
            , '[\s., ]')
      union
     select regexp_split_to_table('Sed ut perspiciatis unde omnis iste natus
        error sit voluptatem accusantium doloremque laudantium, totam rem
        aperiam, eaque ipsa quae ab illo inventore veritatis et quasi
        architecto beatae vitae dicta sunt explicabo. Nemo enim ipsam
        voluptatem quia voluptas sit aspernatur aut odit aut fugit, sed quia
        consequuntur magni dolores eos qui ratione voluptatem sequi
        nesciunt. Neque porro quisquam est, qui dolorem ipsum quia dolor sit
        amet, consectetur, adipisci velit, sed quia non numquam eius modi
        tempora incidunt ut labore et dolore magnam aliquam quaerat
        voluptatem. Ut enim ad minima veniam, quis nostrum exercitationem
        ullam corporis suscipit laboriosam, nisi ut aliquid ex ea commodi
        consequatur? Quis autem vel eum iure reprehenderit qui in ea
        voluptate velit esse quam nihil molestiae consequatur, vel illum qui
        dolorem eum fugiat quo voluptas nulla pariatur?'
            , '[\s., ]')
      union
     select regexp_split_to_table('At vero eos et accusamus et iusto odio
        dignissimos ducimus qui blanditiis praesentium voluptatum deleniti
        atque corrupti quos dolores et quas molestias excepturi sint
        occaecati cupiditate non provident, similique sunt in culpa qui
        officia deserunt mollitia animi, id est laborum et dolorum fuga. Et
        harum quidem rerum facilis est et expedita distinctio. Nam libero
        tempore, cum soluta nobis est eligendi optio cumque nihil impedit
        quo minus id quod maxime placeat facere possimus, omnis voluptas
        assumenda est, omnis dolor repellendus. Temporibus autem quibusdam
        et aut officiis debitis aut rerum necessitatibus saepe eveniet ut et
        voluptates repudiandae sint et molestiae non recusandae. Itaque
        earum rerum hic tenetur a sapiente delectus, ut aut reiciendis
        voluptatibus maiores alias consequatur aut perferendis doloribus
```

```
        asperiores repellat.'
            , '[\s., ]')
)
  insert into sandbox.lorem(word)
       select word
         from w
        where word is not null
          and word <> '';

create or replace function random(a int, b int)
  returns int
  volatile
  language sql
as $$
  select a + ((b-a) * random())::int;
$$;

create or replace function sandbox.lorem(len int)
  returns text
  volatile
  language sql
as $$
  with words(w) as (
      select word
       from sandbox.lorem
   order by random()
      limit len
  )
  select string_agg(w, ' ')
    from words;
$$;

commit;
```

The not-so-random Latin text comes from Lorem Ipsum and is a pretty good base for generating random content. We go even further by separating words from their context and then aggregating them together completely at random in the *sandbox.lorem(int)* function.

The method we use to get N words at random is known to be rather inefficient given large data sources. If you have this use case to solve with a big enough table, then have a look at selecting random rows from a table article from Andrew Gierth, now a PostgreSQL committer.

Modeling Example

Now that we have some data to play with, we can test some application queries for known user stories in the *MVP*, like maybe listing the most recent articles per category with the first three comments on each article.

That's when we realize our previous schema design misses publication timestamps for articles and comments. We need to add this information to our draft model. As it is all a draft with random data, the easiest way around this you already *committed* the data previously (by editing the script) is to simply *drop schema cascade* as shown here:

```
yesql# drop schema sandbox cascade;

NOTICE:  drop cascades to 5 other objects
DETAIL:  drop cascades to table sandbox.lorem
drop cascades to function sandbox.lorem(integer)
drop cascades to table sandbox.category
drop cascades to table sandbox.article
drop cascades to table sandbox.comment
DROP SCHEMA
```

The next version of our schema then looks like this:

```
1  begin;
2
3  create schema if not exists sandbox;
4
5  create table sandbox.category
6   (
7     id    serial primary key,
8     name  text not null
9   );
10
11 insert into sandbox.category(name)
12      values ('sport'),('news'),('box office'),('music');
13
14 create table sandbox.article
15  (
16    id          bigserial primary key,
17    category    integer references sandbox.category(id),
18    pubdate     timestamptz,
19    title       text not null,
20    content     text
21  );
22
23 create table sandbox.comment
24  (
25    id          bigserial primary key,
26    article     integer references sandbox.article(id),
```

```
   pubdate    timestamptz,
   content    text
 );

insert into sandbox.article(category, title, pubdate, content)
     select random(1, 4) as category,
            initcap(sandbox.lorem(5)) as title,
            random( now() - interval '3 months',
                    now() + interval '1 months') as pubdate,
            sandbox.lorem(100) as content
       from generate_series(1, 1000) as t(x);

insert into sandbox.comment(article, pubdate, content)
     select random(1, 1000) as article,
            random( now() - interval '3 months',
                    now() + interval '1 months') as pubdate,
            sandbox.lorem(150) as content
       from generate_series(1, 50000) as t(x);

select article.id, category.name, title
  from      sandbox.article
       join sandbox.category
         on category.id = article.category
 limit 3;

select count(*),
       avg(length(title))::int as avg_title_length,
       avg(length(content))::int as avg_content_length
  from sandbox.article;

   select article.id, article.title, count(*)
     from      sandbox.article
          join sandbox.comment
            on article.id = comment.article
group by article.id
order by count desc
   limit 5;

select category.name,
       count(distinct article.id) as articles,
       count(*) as comments
  from      sandbox.category
       left join sandbox.article on article.category = category.id
       left join sandbox.comment on comment.article = article.id
group by category.name
order by category.name;

commit;
```

To be able to generate random timestamp entries, the script uses another function that's not provided by default in PostgreSQL, and here's its defi-

nition:

```
create or replace function random
 (
  a timestamptz,
  b timestamptz
 )
 returns timestamptz
 volatile
 language sql
as $$
  select a
         + random(0, extract(epoch from (b-a))::int)
           * interval '1 sec';
$$;
```

Now we can have a go at solving the first query of the product's *MVP*, as specified before, on this schema draft version. That should provide a taste of the schema and how well it implements the business rules.

The following query lists the most recent articles per category with the first three comments on each article:

```
\set comments 3
\set articles 1

  select category.name as category,
         article.pubdate,
         title,
         jsonb_pretty(comments) as comments

    from sandbox.category
         /*
          * Classic implementation of a Top-N query
          * to fetch 3 most articles per category
          */
         left join lateral
         (
            select id,
                   title,
                   article.pubdate,
                   jsonb_agg(comment) as comments
              from sandbox.article
                  /*
                   * Classic implementation of a Top-N query
                   * to fetch 3 most recent comments per article
                   */
                  left join lateral
                  (
                      select comment.pubdate,
                             substring(comment.content from 1 for 25) || '…'
```

```
                            as content
                      from sandbox.comment
                     where comment.article = article.id
                  order by comment.pubdate desc
                     limit :comments
                )
                as comment
                on true   -- required with a lateral join

            where category = category.id

         group by article.id
         order by article.pubdate desc
            limit :articles
       )
       as article
       on true -- required with a lateral join

order by category.name, article.pubdate desc;
```

The first thing we notice when running this query is the lack of indexing for it. This chapter contains a more detailed guide on indexing, so for now in the introductory material we just issue these statements:

```
create index on sandbox.article(pubdate);
create index on sandbox.comment(article);
create index on sandbox.comment(pubdate);
```

Here's the query result set, with some content removed. The query has been edited for a nice result text which fits in the book pages, using *jsonb_pretty()* and *substring()*. When embedding it in application's code, this extra processing ougth to be removed from the query. Here's the result, with a single article per category and the three most recent comments per article, as a *JSONB* document:

```
-[ RECORD 1 ]-----------------------------------------------------
category | box office
pubdate  | 2017-09-30 07:06:49.681844+02
title    | Tenetur Quis Consectetur Anim Voluptatem
comments | [                                                      ↵
         |     {                                                  ↵
         |         "content": "adipisci minima ducimus r···",     ↵
         |         "pubdate": "2017-09-27T09:43:24.681844+02:00"↵
         |     },                                                 ↵
         |     {                                                  ↵
         |         "content": "maxime autem modi ex even···",     ↵
         |         "pubdate": "2017-09-26T00:34:51.681844+02:00"↵
         |     },                                                 ↵
         |     {                                                  ↵
         |         "content": "ullam dolorem velit quasi···",     ↵
```

```
          |         "pubdate": "2017-09-25T00:34:57.681844+02:00"↵
          |     }                                                 ↵
          | ]
=[ RECORD 2 ]=========================================================
category | music
pubdate  | 2017-09-28 14:51:13.681844+02
title    | Aliqua Suscipit Beatae A Dolor
...
=[ RECORD 3 ]=========================================================
category | news
pubdate  | 2017-09-30 05:05:51.681844+02
title    | Mollit Omnis Quaerat Do Odit
...
=[ RECORD 4 ]=========================================================
category | sport
pubdate  | 2017-09-29 17:08:13.681844+02
title    | Placeat Eu At Consequuntur Explicabo
...
```

We get this result in about 500ms to 600ms on a laptop, and the timing is down to about 150ms when the *substring(comment.content from 1 for 25) || '...'* part is replaced with just *comment.content*. It's fair to use it in production, with the proper caching strategy in place, i.e. we expect more article reads than writes. You'll find more on caching later in this chapter.

Our schema is a good first version for answering the *MVP*:

- It follows normalization rules as seen in the next parts of this chapter.
- It allows writing the main use case as a single query, and even if the query is on the complex side it runs fast enough with a sample of tens of thousands of articles and fifty thousands of comments.
- The schema allows an easy implementation of workflows for editing categories, articles, and comments.

This draft schema is a SQL file, so it's easy to check it into your versioning system, share it with your colleagues and deploy it to development, integration and continuous testing environments.

For visual schema needs, tools are available that connect to a PostgreSQL database and help in designing a proper set of diagrams from the live schema.

Normalization

Your database model is there to support all your business cases and continuously provide a consistent view of your world as a whole. For that to be possible, some rules have been built up and improved upon over the years. The main goal of those design rules is an overall consistency for all the data managed in your schema.

> *Database normalization is the process of organizing the columns (attributes) and tables (relations) of a relational database to reduce data redundancy and improve data integrity. Normalization is also the process of simplifying the design of a database so that it achieves the optimal structure. It was first proposed by Edgar F. Codd, as an integral part of a relational model.*

Data Structures and Algorithms

After having done all those SQL queries and reviewed *join* operations, *grouping* operations, filtering in the *where* clause and other more sophisticated processing, it should come as no surprise that SQL is declarative, and as such we are not writing the algorithms to execute in order to retrieve the data we need, but rather expressing what is the result set that we are interested into.

Still, PostgreSQL transforms our declarative query into an *execution plan*. This plan makes use of classical algorithms such as *nested loops*, *merge joins*, and *hash joins*, and also in-memory *quicksort* or a *tape sort* when data doesn't fit in memory and PostgreSQL has to spill to disk. The planner and optimiser in PostgreSQL also know how to divide up a single query's work into several concurrent workers for obtaining a result in less time.

When implementing the algorithms ourselves, we know that the most important thing to get right is the data structure onto which we implement computations. As Rob Pike says it in Notes on Programming in C:

> *Rule 5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost al-*

ways be self-evident. Data structures, not algorithms, are central to programming. (See Brooks p. 102.)

In Basics of the Unix Philosophy we read some design principles of the Unix operating system that apply almost verbatim to the problem space of database modeling:

1. *Rule of Modularity*

   Write simple parts connected by clean interfaces.

2. *Rule of Clarity*

   Clarity is better than cleverness.

3. *Rule of Composition*

   Design programs to be connected to other programs.

4. *Rule of Separation*

   Separate policy from mechanism; separate interfaces from engines.

5. *Rule of Simplicity*

   Design for simplicity; add complexity only where you must.

6. *Rule of Parsimony*

   Write a big program only when it is clear by demonstration that nothing else will do.

7. *Rule of Transparency*

   Design for visibility to make inspection and debugging easier.

8. *Rule of Robustness*

   Robustness is the child of transparency and simplicity.

9. *Rule of Representation*

   Fold knowledge into data so program logic can be stupid and robust.

10. *Rule of Least Surprise*

    In interface design, always do the least surprising thing.

11. *Rule of Silence*

When a program has nothing surprising to say, it should say nothing.

12. *Rule of Repair*

    When you must fail, fail noisily and as soon as possible.

13. *Rule of Economy*

    Programmer time is expensive; conserve it in preference to machine time.

14. *Rule of Generation*

    Avoid hand-hacking; write programs to write programs when you can.

15. *Rule of Optimization*

    Prototype before polishing. Get it working before you optimize it.

16. *Rule of Diversity*

    Distrust all claims for "one true way".

17. *Rule of Extensibility*

    Design for the future, because it will be here sooner than you think.

While some of those (such as *rule of silence*) can't really apply to database modeling, most of them do so in a very direct way. Normal forms offer a practical way to enforce respect for those rules. SQL provides a clean interface to connect our data structures: the join operations.

As we're going to see later, a database model with fewer tables isn't a better or simpler data model. The *Rule of Separation* might be the most important in that list. Also, the *Rule of Representaion* in database modeling is reflected directly in the choice of correct data types with advanced behavior and processing function availability.

To summarize all those rules and the different levels for normal forms, I believe that you need to express your *intentions* first. Anyone reading your database schema should instantly understand your business model.

Normal Forms

There are several levels of normalization and the web site dbnormalization.com offers a practical guide to them. In this quick introduction to database normalization, we include the definition of the normal forms:

- 1st Normal Form (*1NF*)

  A table (relation) is in *1NF* if:

  1. There are no duplicated rows in the table.
  2. Each cell is single-valued (no repeating groups or arrays).
  3. Entries in a column (field) are of the same kind.

- 2nd Normal Form (*2NF*)

  A table is in *2NF* if it is in *1NF* and if all non-key attributes are dependent on all of the key. Since a partial dependency occurs when a non-key attribute is dependent on only a part of the composite key, the definition of *2NF* is sometimes phrased as: "A table is in *2NF* if it is in *1NF* and if it has no partial dependencies."

- 3rd Normal Form (*3NF*)

  A table is in *3NF* if it is in *2NF* and if it has no transitive dependencies.

- Boyce-Codd Normal Form (*BCNF*)

  A table is in *BCNF* if it is in *3NF* and if every determinant is a candidate key.

- 4th Normal Form (*4NF*)

  A table is in *4NF* if it is in *BCNF* and if it has no multi-valued dependencies.

- 5th Normal Form (*5NF*)

  A table is in *5NF*, also called "Projection-join Normal Form" (*PJNF*), if it is in *4NF* and if every join dependency in the table is a consequence of the candidate keys of the table.

- Domain-Key Normal Form (*DKNF*)

> A table is in *DKNF* if every constraint on the table is a logical consequence of the definition of keys and domains.

What all of this say is that if you want to be able to process data in your database, using the relational model and SQL as your main tooling, then it's best not to make a total mess of the information and keep it logically structured.

In practice database models often reach for *BCNF* or *4NF*; going all the way to the *DKNF* design is only seen in specific cases.

Database Anomalies

Failure to normalize your model may cause *database anomalies*. Quoting the wikipedia article again:

When an attempt is made to modify (update, insert into, or delete from) a relation, the following undesirable side-effects may arise in relations that have not been sufficiently normalized:

- *Update anomaly*

  The same information can be expressed on multiple rows; therefore updates to the relation may result in logical inconsistencies. For example, each record in an "Employees' Skills" relation might contain an Employee ID, Employee Address, and Skill; thus a change of address for a particular employee may need to be applied to multiple records (one for each skill). If the update is only partially successful — the employee's address is updated on some records but not others — then the relation is left in an inconsistent state. Specifically, the relation provides conflicting answers to the question of what this particular employee's address is. This phenomenon is known as an update anomaly.

- *Insertion anomaly*

  There are circumstances in which certain facts cannot be recorded at all. For example, each record in a "Faculty and Their Courses" relation might contain a Faculty ID, Faculty Name, Faculty Hire Date, and Course Code. Therefore we can record the details of any faculty member who teaches at least one course, but we cannot record a

newly hired faculty member who has not yet been assigned to teach any courses, except by setting the Course Code to null. This phenomenon is known as an insertion anomaly.

- *Deletion anomaly*

  Under certain circumstances, deletion of data representing certain facts necessitates deletion of data representing completely different facts. The "Faculty and Their Courses" relation described in the previous example suffers from this type of anomaly, for if a faculty member temporarily ceases to be assigned to any courses, we must delete the last of the records on which that faculty member appears, effectively also deleting the faculty member, unless we set the Course Code to null. This phenomenon is known as a deletion anomaly.

A database model that implements normal forms avoids those anomalies, and that's why *BCNF* or *4NF* are recommended. Sometimes though some trade-offs are possible with the normalization process, as in the following example.

Modeling an Adress Field

Modeling an address field is a practical use case for normalization, where if you want to respect all the rules you end up with a very complex schema. That said, the answer depends on your application domain; it's not the same if you are connecting people to your telecom network, shipping goods, or just invoicing at the given address.

For invoicing, all we need is a *text* column where to store whatever our user is entering. Our only use for that information is going to be for printing invoices, and we will be sending the invoice in PDF over e-mail anyway.

Now if you're in the delivery business, you need to ensure the address physically exists, is reachable by your agents, and you might need to optimize delivery routes by packing together goods in the same truck and finding the most efficient route in terms of fuel consumption, time spent and how many packages you can deliver in a single shift.

Then an address field looks quite different than a single *text* entry:

- We need to have a — possibly geolocalized — list of cities as a reference, and we know that the same city name can be found in several regions, such as Portland which is a very common name apparently.
- So for our cities, we need a reference table of districts and regions within each country (regions would be states in the USA, Länder in Germany, etc), and then it's possible to reference a city without ambiguity.
- Each city is composed of a list of streets, and of course, those names are reused a lot within cities of regions using the same language, so we need a reference table of street names and then an association table of street names found in cities.
- We then need a number for the street, and depending on the city the same street name will not host the same numbers, so that's information relevant for the association of a city and a street.
- Each number on the street might have to be geo-localized with precision, depending on the specifics of your business.
- Also, if we run a business that delivers to the door (and for example assembles furniture, or connects electricity or internet to people homes), we need per house and per-apartment information for each number in a specific street.
- Finally, our users might want to refer to their place by *zip code*, although a postal code might cover a district or an area within a city, or group several cities, usually small rural communities.

A database model that is still simple to enable delivery to known places would then involve at least the five following tables, written in pseudo SQL (meaning that this code won't actually run):

```
create table country(code, name);
create table region(country, name);
create table city(country, region, name, zipcode);
create table street(name);
create table city_street_numbers
       (country, region, city, street, number, location);
```

Then it's possible to implement an advanced input form with normalization of the delivery address and to compute routes. Again, if all you're doing with the address is printing it on PDF documents (contracts, invoices,

etc.) and sometimes to an envelope label, you might not need to be this sophisticated.

In the case of the addresses, it's important to then implement a maintenance process for all those countries, regions and cities where your business operates. Borders are evolving in the world, and you might need to react to those changes. Postal codes usually change depending on population counts, so there again you need to react to such changes. Moreover streets get renamed, and new streets are constructed. New buildings are built and sometimes given new numbers such as *2 bis* or *4 ter*. So even the number information isn't an *integer* field...

The point of a proper data model is to make it easy for the application to process the information it needs, and to ensure global consistency for the information. The address exercise doesn't allow for understanding of those points, and we've reached its limits already.

Primary Keys

Primary keys are a database constraint allowing us to implement the first and second normal forms. The first rule to follow to reach first normal form says *"There are no duplicated rows in the table"*.

A primary key ensures two things:

- The attributes that are part of the *primary key* constraint definition are not allowed to be *null*.
- The attributes that are part of the *primary key* are unique in the table's content.

To ensure that there is no duplicated row, we need the two guarantees. Comparing *null* values in SQL is a complex matter, and rather than argue if the no-duplicate rule applies to *null* = *null* (which is *null*) or to *null is not null* (which is false), a *primary key* constraint disallow *null* values entirely.

Surrogate Keys

The reason why we have *primary key* is to avoid duplicate entries in the data set. As soon as a *primary key* is defined on an automatically generated column, which is arguably not really part of the data set, then we open the gates for violation of the first normal form.

Earlier in this chapter, we drafted a database model with the following table:

```
create table sandbox.article
 (
   id          bigserial primary key,
   category    integer references sandbox.category(id),
   pubdate     timestamptz,
   title       text not null,
   content     text
 );
```

This model isn't even compliant with *1NF*:

```
insert into sandbox.article (category, pubdate, title)
     values (2, now(), 'Hot from the Press'),
            (2, now(), 'Hot from the Press')
  returning *;
```

PostgreSQL is happy to insert duplicate entries here:

```
-[ RECORD 1 ]──────────────────────────
id       | 1001
category | 2
pubdate  | 2017-08-30 18:09:46.997924+02
title    | Hot from the Press
content  | ¤
=[ RECORD 2 ]══════════════════════════
id       | 1002
category | 2
pubdate  | 2017-08-30 18:09:46.997924+02
title    | Hot from the Press
content  | ¤

INSERT 0 2
```

Of course, it's possible to argue that those entries are not duplicates: they each have their own *id* value, which is different — and it is an artificial value derived automatically for us by the system.

Actually, we now have to deal with two article entries in our publication system with the same category (category 2 is *news*), the same title, and

the same publication date. I don't suppose this is an acceptable situation for the business rules.

In term of database modeling, the artificially generated key is named a *surrogate key* because it is a substitute for a *natural key*. A *natural key* would allow preventing duplicate entries in our data set.

We can fix our schema to prevent duplicate entries:

```
create table sandbox.article
 (
   category   integer references sandbox.category(id),
   pubdate    timestamptz,
   title      text not null,
   content    text,

   primary key(category, title);
 );
```

Now, you can share the same article's title in different categories, but you can only publish with a title once in the whole history of our publication system. Given this alternative design, we allow publications with the same title at different publication dates. It might be needed, after all, as we know that history often repeats itself.

```
create table sandboxpk.article
 (
   category   integer references sandbox.category(id),
   pubdate    timestamptz,
   title      text not null,
   content    text,

   primary key(category, pubdate, title)
 );
```

Say we go with the solution allowing reusing the same title at a later date. We now have to change the model of our *comment* table, which references the *sandbox.article* table:

```
create table sandboxpk.comment
 (
   a_category integer     not null,
   a_pubdate  timestamptz not null,
   a_title    text        not null,
   pubdate    timestamptz,
   content    text,

   primary key(a_category, a_pubdate, a_title, pubdate, content),

```

```
    foreign key(a_category, a_pubdate, a_title)
     references sandboxpk.article(category, pubdate, title)
);
```

As you can see each entry in the *comment* table must have enough information to be able to reference a single entry in the *article* table, with a guarantee that there are no duplicates.

We then have quite a big table for the data we want to manage in there. So there's yet another solution to this *surrogate* key approach, a trade-off where you have the generated summary key benefits and still the natural primary key guarantees needed for the *1NF*:

```
create table sandboxpk.article
(
  id          bigserial primary key,
  category    integer      not null references sandbox.category(id),
  pubdate     timestamptz  not null,
  title       text         not null,
  content     text,

  unique(category, pubdate, title)
);
```

Now the *category*, *pubdate* and *title* have a *not null* constraint and a *unique* constraint, which is the same level of guarantee as when declaring them a *primary key*. So we both have a *surrogate* key that's easy to reference from other tables in our model, and also a strong *1NF* guarantee about our data set.

Foreign Keys Constraints

Proper *primary keys* allow implementing *1NF*. Better normalization forms are achieved when your data model is clean: any information is managed in a single place, which is a single source of truth. Then, your data has to be split into separate tables, and that's when other constraints are needed.

To ensure that the information still makes sense when found in different tables, we need to be able to *reference* information and ensure that our *reference* keeps being valid. That's implemented with a *foreign key* constraint.

A *foreign key* constraint must reference a set of keys known to be *unique* in the target table, so PostgreSQL enforce the presence of either a *unique* or

a *primary key* constraint on the target table. Such a constraint is always implemented in PostgreSQL with a *unique* index. PostgreSQL doesn't create indexes at the source side of the *foreign key* constraint, though. If you need such an index, you have to explicitly create it.

Not Null Constraints

The *not null* constraint disallows unspecified entries in attributes, and the data type of the attribute forces its value to make sense, so the data type can also be considered to be kind of constraint.

Check Constraints and Domains

When the data type allows more values than your application or business model, SQL allows you to restrict the values using either a *domain* definition or a *check* constraint. The domain definition applies a *check* constraint to a data type definition. Here's the example from the PostgreSQL documentation chapter about check constraints:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0)
);
```

The *check* constraint can also reference several columns of the same table at once, if that's required:

```
CREATE TABLE products (
    product_no integer,
    name text,
    price numeric CHECK (price > 0),
    discounted_price numeric,
    CHECK (discounted_price > 0 AND price > discounted_price)
);
```

And here's how to define a new data domain as per the PostgreSQL documentation for the CREATE DOMAIN SQL command:

```
CREATE DOMAIN us_postal_code AS TEXT
 CHECK
 (
   VALUE ~ '^\d{5}$'
    OR
```

```
    VALUE ~ '^\d{5}-\d{4}$'
);
```

It is now possible to use this domain definition as a data type, as in the following example from the same documentation page:

```
CREATE TABLE us_snail_addy (
  address_id SERIAL PRIMARY KEY,
  street1 TEXT NOT NULL,
  street2 TEXT,
  street3 TEXT,
  city TEXT NOT NULL,
  postal us_postal_code NOT NULL
);
```

Exclusion Constraints

As seen in the presentation of Ranges in the previous chapter, it's also possible to define *exclusion constraints* with PostgreSQL. Those work like a generalized *unique* constraint, with a custom operator choice. The example we used is the following, where an exchange rate is valid for a period of time and we do not allow overlapping periods of validity for a given rate:

```
create table rates
 (
  currency text,
  validity daterange,
  rate     numeric,

  exclude using gist (currency with =,
                      validity with &&)
 );
```

Practical Use Case: Geonames

> *The GeoNames geographical database covers all countries and contains over eleven million place names that are available for download free of charge.*

The website offers online querying and all the data is made available to download and use. As is often the case, it comes in an ad-hoc format

and requires some processing and normalization before it's usable in a PostgreSQL database.

```
begin;

create schema if not exists raw;

create table raw.geonames
 (
   geonameid         bigint,
   name              text,
   asciiname         text,
   alternatenames    text,
   latitude          double precision,
   longitude         double precision,
   feature_class     text,
   feature_code      text,
   country_code      text,
   cc2               text,
   admin1_code       text,
   admin2_code       text,
   admin3_code       text,
   admin4_code       text,
   population        bigint,
   elevation         bigint,
   dem               bigint,
   timezone          text,
   modification      date
 );

create table raw.country
 (
  iso                text,
  iso3               text,
  isocode            integer,
  fips               text,
  name               text,
  capital            text,
  area               double precision,
  population         bigint,
  continent          text,
  tld                text,
  currency_code      text,
  currency_name      text,
  phone              text,
  postal_code_format text,
  postal_code_regex  text,
  languages          text,
  geonameid          bigint,
  neighbours         text,
  fips_equiv         text
 );
```

```
50
51 \copy raw.country from 'countryInfoData.txt' with csv delimiter E'\t'
52
53 create table raw.feature
54  (
55   code        text,
56   description text,
57   comment     text
58  );
59
60 \copy raw.feature from 'featureCodes_en.txt' with csv delimiter E'\t'
61
62 create table raw.admin1
63  (
64   code       text,
65   name       text,
66   ascii_name text,
67   geonameid  bigint
68  );
69
70 \copy raw.admin1 from 'admin1CodesASCII.txt' with csv delimiter E'\t'
71
72 create table raw.admin2
73  (
74   code       text,
75   name       text,
76   ascii_name text,
77   geonameid  bigint
78  );
79
80 \copy raw.admin2 from 'admin2Codes.txt' with csv delimiter E'\t'
81
82 commit;
```

Once we have loaded the raw data from the published files at http://download.geonames.org/export/dump/, we can normalize the content and begin to use the data.

You might notice that the SQL file above is missing the `\copy` command for the *raw.geonames* table. That's because *copy* failed to load the file properly: some location names include single and double quotes, and those are not properly quoted... and not properly escaped. So we resorted to pgloader to load the file, with the following command:

```
load csv
  from /tmp/geonames/allCountries.txt
  into pgsql://appdev@/appdev
  target table raw.geonames

  with fields terminated by '\t',
       fields optionally enclosed by '§',
```

```
       fields escaped by '%',
       truncate;
```

Here's the summary obtained when loading the dataset on the laptop used to prepare this book:

```
             table name     errors       rows      bytes      total time
-----------------------  ---------  ---------  ---------  --------------
                  fetch          0          0                    0.009s
-----------------------  ---------  ---------  ---------  --------------
          raw.geonames          0   11540466     1.5 GB       6m43.218s
-----------------------  ---------  ---------  ---------  --------------
        Files Processed          0          1                    0.026s
COPY Threads Completion          0          2                   6m43.319s
-----------------------  ---------  ---------  ---------  --------------
      Total import time          ✓          3     1.5 GB       6m43.345s
```

To normalize the schema, we apply the rules from the definition of the *normal forms* as seen previously. Basically, we want to avoid any dependency in between the attributes of our models. Any dependency means that we need to create a separate table where to manage a set of data that makes sense in isolation is managed.

The *raw.geonames* table uses several reference data that *GeoNames* provide as separate downloads. We then need to begin with fixing the reference data used in the model.

Features

The *GeoNames* model tags all of its geolocation data with a *feature* class and a feature. The description for those codes are detailed on the GeoNames codes page and available for download in the *featureCodes_en.txt* file. Some of the information we need is only available in a text form and has to be reported manually.

```
1  begin;
2
3  create schema if not exists geoname;
4
5  create table geoname.class
6   (
7    class        char(1) not null primary key,
8    description  text
9   );
10
11 insert into geoname.class (class, description)
12      values ('A', 'country, state, region,...'),
```

```
            ('H', 'stream, lake, ...'),
            ('L', 'parks,area, ...'),
            ('P', 'city, village,...'),
            ('R', 'road, railroad '),
            ('S', 'spot, building, farm'),
            ('T', 'mountain,hill,rock,... '),
            ('U', 'undersea'),
            ('V', 'forest,heath,...');

create table geoname.feature
 (
  class       char(1) not null references geoname.class(class),
  feature     text    not null,
  description text,
  comment     text,

  primary key(class, feature)
 );

insert into geoname.feature
     select substring(code from 1 for 1) as class,
            substring(code from 3) as feature,
            description,
            comment
       from raw.feature
      where feature.code <> 'null';

commit;
```

As we see in this file we have to deal with an explicit *'null'* entry: there's a text that is four letters long in the last line (and reads `null`) and that we don't want to load.

Also, the provided file uses the notation *A.ADM1* for an entry of class *A* and feature *ADM1*, which we split into proper attributes in our normalization process. The natural key for the *geoname.feature* table is the combination of the *class* and the *feature*.

Once all the data is loaded and normalized, we can get some nice statistics:

```
  select class, feature, description, count(*)
    from feature
         left join geoname using(class,feature)
group by class, feature
order by count desc
   limit 10;
```

This is a very simple top-10 query, per feature:

```
 class | feature |   description    | count
```

```
═══════╪═══════════╪═══════════════════╪═══════════
 P     │ PPL       │ populated place   │ 1711458
 H     │ STM       │ stream            │  300283
 S     │ CH        │ church            │  236394
 S     │ FRM       │ farm              │  234536
 S     │ SCH       │ school            │  223402
 T     │ HLL       │ hill              │  212659
 T     │ MT        │ mountain          │  192454
 S     │ HTL       │ hotel             │  170896
 H     │ LK        │ lake              │  162922
 S     │ BLDG      │ building(s)       │  143742
(10 rows)
```

Countries

The *raw.country* table has several normalization issues. Before we list them, having a look at some data will help us:

```
-[ RECORD 1 ]──────┬──────────────────────────
iso                │ FR
iso3               │ FRA
isocode            │ 250
fips               │ FR
name               │ France
capital            │ Paris
area               │ 547030
population         │ 64768389
continent          │ EU
tld                │ .fr
currency_code      │ EUR
currency_name      │ Euro
phone              │ 33
postal_code_format │ #####
postal_code_regex  │ ^(\d{5})$
languages          │ fr-FR,frp,br,co,ca,eu,oc
geonameid          │ 3017382
neighbours         │ CH,DE,BE,LU,IT,AD,MC,ES
fips_equiv         │ ¤
```

The main normalization failures we see are:

- Nothing guarantees the absence of duplicate rows in the table, so we need to add a *primary key* constraint.

  Here the *isocode* attribute looks like the best choice, as it's both unique and an integer.

- The *languages* and *neighbours* attributes both contain multiple-valued content, a comma-separated list of either languages or country codes.
- To reach *2NF* then, all non-key attributes should be dependent on the entire of the key, and the currencies and postal code formats are not dependent on the country.

A good way to check for dependencies on the key attributes is with the following type of query:

```
  select currency_code, currency_name, count(*)
    from raw.country
group by currency_code, currency_name
order by count desc
   limit 5;
```

In our dataset, we have the following result, showing 34 countries using the Euro currency:

```
 currency_code | currency_name | count
═══════════════╪═══════════════╪═══════
 EUR           | Euro          |    34
 USD           | Dollar        |    16
 AUD           | Dollar        |     8
 XOF           | Franc         |     8
 XCD           | Dollar        |     8
(5 rows)
```

In this book, we're going to pass on the currency, language, and postal code formats of countries and focus on some information only. That gives us the following normalization process:

```
begin;

create schema if not exists geoname;

create table geoname.continent
 (
  code    char(2) primary key,
  name    text
 );

insert into geoname.continent(code, name)
     values ('AF', 'Africa'),
            ('NA', 'North America'),
            ('OC', 'Oceania'),
            ('AN', 'Antarctica'),
            ('AS', 'Asia'),
            ('EU', 'Europe'),
```

```
            ('SA', 'South America');

create table geoname.country
 (
  isocode   integer primary key,
  iso       char(2) not null,
  iso3      char(3) not null,
  fips      text,
  name      text,
  capital   text,
  continent char(2) references geoname.continent(code),
  tld       text,
  geonameid bigint
 );

insert into geoname.country
     select isocode, iso, iso3, fips, name,
            capital, continent, tld, geonameid
       from raw.country;

create table geoname.neighbour
 (
  isocode   integer not null references geoname.country(isocode),
  neighbour integer not null references geoname.country(isocode),

  primary key(isocode, neighbour)
 );

insert into geoname.neighbour
   with n as(
     select isocode,
            regexp_split_to_table(neighbours, ',') as neighbour
       from raw.country
   )
   select n.isocode,
          country.isocode
     from n
          join geoname.country
            on country.iso = n.neighbour;

commit;
```

Note that we add the continent list (for completeness in the region drill down) and then introduce the *geoname.neighbour* part of the model. Having an association table that *links* every country with its neighbours on the map (a neighbour has a common border) allows us to easily query for the information:

```
select neighbour.iso,
       neighbour.name,
       neighbour.capital,
```

```
4        neighbour.tld
5
6    from geoname.neighbour as border
7
8        join geoname.country as country
9          on border.isocode = country.isocode
10
11       join geoname.country as neighbour
12         on border.neighbour = neighbour.isocode
13
14   where country.iso = 'FR';
```

So we get the following list of neighbor countries for France:

```
 iso │    name     │     capital      │ tld
═════╪═════════════╪══════════════════╪═════
 CH  │ Switzerland │ Bern             │ .ch
 DE  │ Germany     │ Berlin           │ .de
 BE  │ Belgium     │ Brussels         │ .be
 LU  │ Luxembourg  │ Luxembourg       │ .lu
 IT  │ Italy       │ Rome             │ .it
 AD  │ Andorra     │ Andorra la Vella │ .ad
 MC  │ Monaco      │ Monaco           │ .mc
 ES  │ Spain       │ Madrid           │ .es
(8 rows)
```

Administrative Zoning

The raw data from the *GeoNames* website then offers an interesting geographical breakdown in the *country_code*, *admin1_code* and *admin2_code*.

```
1  select geonameid, name, admin1_code, admin2_code
2    from raw.geonames
3   where country_code = 'FR'
4   limit 5
5  offset 50;
```

To get an interesting result set, we select randomly from the data for France, where the code has to be expanded to be meaningful. With a USA based data set, we get states codes as *admin1_code* (e.g. *IL* for Illinois), and the necessity for normalized data might then be less visible.

Of course, never use *offset* in your application queries, as seen previously. Here, we are doing interactive discovery of the data, so it is found acceptable, to some extent, to play with the *offset* facility.

Here's the data set we get:

```
 geonameid |        name         | admin1_code | admin2_code
═══════════╪═════════════════════╪═════════════╪═════════════
   2967132 | Zintzel du Nord     | 44          | 67
   2967133 | Zinswiller          | 44          | 67
   2967134 | Ruisseau de Zingajo | 94          | 2B
   2967135 | Zincourt            | 44          | 88
   2967136 | Zimming             | 44          | 57
(5 rows)
```

The *GeoNames* website provides files *admin1CodesASCII.txt* and *admin2Codes.txt* for us to use to normalize our data. Those files again use admin codes spelled as *AD.06* and *AF.01.1125426* where the *raw.geonames* table uses them as separate fields. That's a good reason to split them now.

Here's the SQL to normalize the admin breakdowns, splitting the codes and adding necessary constraints, to ensure data quality:

```
1  begin;
2
3  create schema if not exists geoname;
4
5  create table geoname.region
6   (
7    isocode   integer not null references geoname.country(isocode),
8    regcode   text not null,
9    name      text,
10   geonameid bigint,
11
12   primary key(isocode, regcode)
13  );
14
15 insert into geoname.region
16    with admin as
17    (
18      select regexp_split_to_array(code, '[.]') as code,
19             name,
20             geonameid
21        from raw.admin1
22    )
23    select country.isocode as isocode,
24           code[2] as regcode,
25           admin.name,
26           admin.geonameid
27      from admin
28           join geoname.country
29             on country.iso = code[1];
30
31 create table geoname.district
32  (
```

```
  isocode   integer not null,
  regcode   text not null,
  discode   text not null,
  name      text,
  geonameid bigint,

  primary key(isocode, regcode, discode),
  foreign key(isocode, regcode)
   references geoname.region(isocode, regcode)
 );

insert into geoname.district
   with admin as
   (
     select regexp_split_to_array(code, '[.]') as code,
            name,
            geonameid
       from raw.admin2
   )
     select region.isocode,
            region.regcode,
            code[3],
            admin.name,
            admin.geonameid
       from admin

            join geoname.country
              on country.iso = code[1]

            join geoname.region
              on region.isocode = country.isocode
             and region.regcode = code[2];

commit;
```

The previous query can now be rewritten, showing region and *district* names rather than *admin1_code* and *admin2_code*, which we still have internally in case we need them of course.

```
select r.name, reg.name as region, d.name as district
  from raw.geonames r

       left join geoname.country
              on country.iso = r.country_code

       left join geoname.region reg
              on reg.isocode = country.isocode
             and reg.regcode = r.admin1_code

       left join geoname.district d
              on d.isocode = country.isocode
```

```
13 |             and d.regcode = r.admin1_code
14 |             and d.discode = r.admin2_code
15 |  where country_code = 'FR'
16 |  limit 5
17 |offset 50;
```

The query uses *left join* operations because we have geo-location data without the *admin1* or *admin2* levels of details — more on that later. Here's the same list of French areas, this time with proper names:

```
        name          |  region   |           district
══════════════════════╪═══════════╪═══════════════════════════
 Zintzel du Nord      | Grand Est | Département du Bas-Rhin
 Zinswiller           | Grand Est | Département du Bas-Rhin
 Ruisseau de Zingajo  | Corsica   | Département de la Haute-Corse
 Zincourt             | Grand Est | Département des Vosges
 Zimming              | Grand Est | Département de la Moselle
(5 rows)
```

Geolocation Data

Now that we have loaded the reference data, we can load the main geolocation data with the following script. Note that we skip parts of the data we don't need for this book, but that you might want to load in your application's background data.

Before loading the raw data into a normalized version of the table, which will make heavy use of the references we normalized before, we have to study and understand how the breakdown works:

```
1 |select count(*) as all,
2 |       count(*) filter(where country_code is null) as no_country,
3 |       count(*) filter(where admin1_code is null) as no_region,
4 |       count(*) filter(where admin2_code is null) as no_district,
5 |       count(*) filter(where feature_class is null) as no_class,
6 |       count(*) filter(where feature_code is null) as no_feat
7 |  from raw.geonames ;
```

We have lots of entries without reference for a *country*, and even more without detailed breakdown (*admin1_code* and *admin2_code* are not always part of the data). Moreover we also have points without any reference feature and class, some of them in the Artic.

```
   all    | no_country | no_region | no_district | no_class | no_feat
══════════╪════════════╪═══════════╪═════════════╪══════════╪═════════
 11540466 |       5821 |     45819 |     5528455 |     5074 |   95368
```

```
(1 row)
```

Given that, our normalization query must be careful to use *left join* operations, so as to allow for fields to be *null* when the foreign key reference doesn't exist. Be careful to drill down properly to the country, then the region, and only then the district, as the data set contains points of several layers of precision as seen in the query above.

```
begin;

create table geoname.geoname
 (
   geonameid         bigint primary key,
   name              text,
   location          point,
   isocode           integer,
   regcode           text,
   discode           text,
   class             char(1),
   feature           text,
   population        bigint,
   elevation         bigint,
   timezone          text,

   foreign key(isocode)
    references geoname.country(isocode),

   foreign key(isocode, regcode)
    references geoname.region(isocode, regcode),

   foreign key(isocode, regcode, discode)
    references geoname.district(isocode, regcode, discode),

   foreign key(class)
    references geoname.class(class),

   foreign key(class, feature)
    references geoname.feature(class, feature)
 );

insert into geoname.geoname
  with geo as
  (
     select geonameid,
            name,
            point(longitude, latitude) as location,
            country_code,
            admin1_code,
            admin2_code,
            feature_class,
            feature_code,
```

```
            population,
            elevation,
            timezone
       from raw.geonames
   )
     select geo.geonameid,
            geo.name,
            geo.location,
            country.isocode,
            region.regcode,
            district.discode,
            feature.class,
            feature.feature,
            population,
            elevation,
            timezone
       from geo
            left join geoname.country
              on country.iso = geo.country_code

            left join geoname.region
              on region.isocode = country.isocode
             and region.regcode = geo.admin1_code

            left join geoname.district
              on district.isocode = country.isocode
             and district.regcode = geo.admin1_code
             and district.discode = geo.admin2_code

           left join geoname.feature
             on feature.class = geo.feature_class
            and feature.feature = geo.feature_code;

create index on geoname.geoname using gist(location);

commit;
```

Now that we have a proper data set loaded, it's easier to make sense of the administrative breakdowns and the geo-location data.

The real use case for this data comes later: thanks to the *GiST* index over the *geoname.location* column we are now fully equipped to do a names lookup from the geo-localized information.

```
  select continent.name,
         count(*),
         round(100.0 * count(*) / sum(count(*)) over(), 2) as pct,
         repeat('■', (100 * count(*) / sum(count(*)) over())::int) as hist
    from geoname.geoname
         join geoname.country using(isocode)
         join geoname.continent
```

```
8 |           on continent.code = country.continent
9 | group by continent.name order by continent.name;
```

We can see that the *GeoNames* data is highly skewed towards Asia, North America, and then Europe. Of course, the Antartica data is not very dense.

```
     name      |  count  |  pct  |                hist
═══════════════╪═════════╪═══════╪═══════════════════════════════════
 Africa        | 1170043 | 10.14 | ██████████
 Antarctica    |   21125 |  0.18 |
 Asia          | 3772195 | 32.70 | ████████████████████████████████
 Europe        | 2488807 | 21.58 | ██████████████████████
 North America | 3210802 | 27.84 | ████████████████████████████
 Oceania       |  354325 |  3.07 | ███
 South America |  517347 |  4.49 | ████
(7 rows)
```

Geolocation GiST Indexing

The previous *geoname* table creation script contains the following index definition:

```
1 | create index on geoname.geoname using gist(location);
```

Such an index is useful when searching for a specific location within our table, which contains about 11.5 million entries. PostgreSQL supports *index scan* based lookups in several situations, including the *kNN* lookup, also known as the *nearest neighbor* lookup.

In the arrays non-relational data type example we loaded a data set of 200,000 geo-localized tweets in the *hashtag* table. Here's an extract of this table's content:

```
-[ RECORD 1 ]──────────────────────────────────────────────────
id       | 720553458596757504
date     | 2016-04-14 10:05:00+02
uname    | Police Calls 32801
message  | #DrugViolation at 335 N Magnolia Ave. #orlpol #opd
location | (-81.3769794,28.5469591)
hashtags | {#DrugViolation,#opd,#orlpol}
```

It's possible to retrieve more information from the *GeoNames* data thanks to the following *lateral left join* lookup in which we implement a *kNN* search with `order by ... <-> ... limit k` clause:

```
1 |   select id,
2 |          round((hashtag.location <-> geoname.location)::numeric, 3) as dist,
```

```
3           country.iso,
4           region.name as region,
5           district.name as district
6       from hashtag
7           left join lateral
8           (
9              select geonameid, isocode, regcode, discode, location
10               from geoname.geoname
11           order by location <-> hashtag.location
12              limit 1
13          )
14          as geoname
15          on true
16          left join geoname.country using(isocode)
17          left join geoname.region using(isocode, regcode)
18          left join geoname.district using(isocode, regcode, discode)
19  order by id
20     limit 5;
```

The `<->` operator computes the distance in between its argument, and by using the *limit 1* clause we select the nearest known entry in the *geoname.geoname* table for each entry in the *hashtag* table.

Then it's easy to add our normalized *GeoNames* information from the *country*, *region* and *district* tables. Here's the result we get here:

```
         id          | dist  | iso |    region    |      district
═════════════════════╪═══════╪═════╪══════════════╪═══════════════════
 720553447402160128  | 0.004 | US  | Florida      | Orange County
 720553457015324672  | 0.004 | US  | Texas        | Smith County
 720553458596757504  | 0.001 | US  | Florida      | Orange County
 720553466804989952  | 0.001 | US  | Pennsylvania | Philadelphia County
 720553475923271680  | 0.000 | US  | New York     | Nassau County
(5 rows)
```

To check that our *GiST* index is actually used, we use the *explain* command of PostgreSQL, with the spelling `explain (costs off)` followed by the whole query as above, and we get the following query plan:

```
\pset format wrapped
\pset columns 70

                              QUERY PLAN
═══════════════════════════════════════════════════════════════════════
 Limit
   ->  Nested Loop Left Join
         ->  Nested Loop Left Join
               ->  Nested Loop Left Join
                     Join Filter: (geoname.isocode = country.isocode)
                     ->  Nested Loop Left Join
                           ->  Index Scan using hashtag_pkey on hasht···
···ag
```

```
                          ->  Limit
                                ->  Index Scan using geoname_locatio···
···n_idx on geoname
                                      Order By: (location <-> hashta···
···g.location)
                    ->  Materialize
                          ->  Seq Scan on country
              ->  Index Scan using region_pkey on region
                    Index Cond: ((geoname.isocode = isocode) AND (ge···
···oname.regcode = regcode))
         ->  Index Scan using district_pkey on district
               Index Cond: ((geoname.isocode = isocode) AND (geoname.···
···regcode = regcode) AND (geoname.discode = discode))
(16 rows)
```

The *index scan using geoname_location_idx on geoname* is clear: the index has been used. On the laptop on which this book has been written, we get the result in about 13 milliseconds.

A Sampling of Countries

This dataset of more than 11 million rows is not practical to include in the book's material for the *Full Edition* and *Enterprise Edition*, where you have a database dump or Docker image to play with. We instead take a random sample of 1% of the table's content, and here's how the magic is done:

```
1  begin;
2
3  create schema if not exists sample;
4
5  drop table if exists sample.geonames;
6
7  create table sample.geonames
8     as select /*
9                * We restrict the "export" to some columns only, so as to
10               * further reduce the size of the exported file available to
11               * download with the book.
12               */
13              geonameid,
14              name,
15              longitude,
16              latitude,
17              feature_class,
18              feature_code,
19              country_code,
20              admin1_code,
21              admin2_code,
22              population,
```

```
            elevation,
            timezone
            /*
             * We only keep 1% of the 11 millions rows here.
             */
        from raw.geonames TABLESAMPLE bernoulli(1);

\copy sample.geonames to 'allCountries.sample.copy'

commit;
```

In this script, we use the *tablesample* feature of PostgreSQL to only keep a random selection of 1% of the rows in the table. The *tablesample* accepts several methods, and you can see the PostgreSQL documentation entitled Writing A Table Sampling Method yourself if you need to.

Here's what the from clause documentation of the *select* statement has to say about the choice of *bernouilli* and *system*, included by default in PostgreSQL:

> The BERNOULLI and SYSTEM sampling methods each accept a single argument which is the fraction of the table to sample, expressed as a percentage between 0 and 100. This argument can be any real-valued expression. (Other sampling methods might accept more or different arguments.) These two methods each return a randomly-chosen sample of the table that will contain approximately the specified percentage of the table's rows. The BERNOULLI method scans the whole table and selects or ignores individual rows independently with the specified probability. The SYSTEM method does block-level sampling with each block having the specified chance of being selected; all rows in each selected block are returned. The SYSTEM method is significantly faster than the BERNOULLI method when small sampling percentages are specified, but it may return a less-random sample of the table as a result of clustering effects.

Running the script, here's what we get:

```
yesql# \i geonames.sample.sql
BEGIN
CREATE SCHEMA
DROP TABLE
SELECT 115904
COPY 115904
COMMIT
```

Our *sample.geonames* table only contains 115,904 rows. Another run of the same query instead yielded 115,071 instead. After all the sampling is made following a random-based algorithm.

Modelization Anti-Patterns

Failures to follow normalization forms opens the door to anomalies as seen previously. Some failure modes are so common in the wild that we can talk about *anti-patterns*. One of the worst possible design choices would be the *EAV* model.

Entity Attribute Values

The entity attribue values or *EAV* is a design that tries to accommodate with a lack of specifications. In our application, we have to deal with parameters and new parameters may be added at each release. It's not clear which parameters we need, we just want a place to manage them easily, and we are already using a database server after all. So there we go:

```
1  begin;
2
3  create schema if not exists eav;
4
5  create table eav.params
6   (
7     entity    text not null,
8     parameter text not null,
9     value     text not null,
10
11    primary key(entity, parameter)
12  );
13
14 commit;
```

You might have already seen this model or a variation of it in the field. The model makes it very easy to add *things* to it, and very difficult to make sense of the accumulated data, or to use them effectively in SQL, making it an anti-pattern.

```
insert into eav.params(entity, parameter, value)
     values ('backend', 'log_level', 'notice'),
            ('backend', 'loglevel', 'info'),
            ('api', 'timeout', '30'),
            ('api', 'timout', '40'),
            ('gold', 'response time', '60'),
            ('gold', 'escalation time', '90'),
            ('platinum', 'response time', '15'),
            ('platinum', 'escalation time', '30');
```

In this example we made some typos on purpose, to show the limits of the *EAV* model. It's impossible to catch those errors, and you might have parts of your code that query one spelling or a different one.

Main problems of this *EAV* anti-pattern are:

- The *value* attribute is of type *text* so as to be able to host about anything, where some parameters are going to be *integer*, *interval*, *inet* or *boolean* values.
- The *entity* and *parameter* fields are likewise free-text, meaning that any typo will actually create new entries, which might not even be used anywhere in the application.
- When fetching all the parameters of an entity to set up your application's object, the parameter names are a value in each row rather than the name of the column where to find them, meaning extra work and loops.
- When you need to process parameter in SQL queries, you need to add a join to the *params* table for each parameter you are interested in.

As an example of the last point, here's a query that fetches the *response time* and the *escalated time* for support customers when using the previous *params* setup. First, we need a quick design for a customer and a support contract table:

```
begin;

create table eav.support_contract_type
 (
   id   serial primary key,
   name text not null
 );

insert into eav.support_contract_type(name)
```

```
    values ('gold'), ('platinum');

create table eav.support_contract
 (
   id       serial primary key,
   type     integer not null references eav.support_contract_type(id),
   validity daterange not null,
   contract text,

   exclude using gist(type with =, validity with &&)
 );

create table eav.customer
 (
   id       serial primary key,
   name     text not null,
   address  text
 );

create table eav.support
 (
   customer  integer not null,
   contract  integer not null references eav.support_contract(id),
   instances integer not null,

   primary key(customer, contract),
   check(instances > 0)
 );

commit;
```

And now it's possible to get customer support contract parameters such as *response time* and *escalation time*, each with its own join:

```
select customer.id,
       customer.name,
       ctype.name,
       rtime.value::interval as "resp. time",
       etime.value::interval as "esc. time"
  from eav.customer
       join eav.support
         on support.customer = customer.id

       join eav.support_contract as contract
         on support.contract = contract.id

       join eav.support_contract_type as ctype
         on ctype.id = contract.type

       join eav.params as rtime
         on rtime.entity = ctype.name
        and rtime.parameter = 'response time'
```

```
19
20        join eav.params as etime
21          on etime.entity = ctype.name
22         and etime.parameter = 'escalation time';
```

Each parameter you add has to be added as an extra *join* operation in the previous query. Also, if someone enters a value for *response time* that isn't compatible with the *interval* data type representation, then the query fails.

Never implement an *EAV* model, this anti-pattern makes everything more complex than it should for a very small gain at modeling time.

It might be that the business case your application is solving actually has an *attribute volatility* problem to solve. In that case, consider having as solid a model as possible and use *jsonb* columns as extension points.

Multiple Values per Column

As seen earlier, a table (relation) is in *1NF* if:

1. There are no duplicated rows in the table.
2. Each cell is single-valued (no repeating groups or arrays).
3. Entries in a column (field) are of the same kind.

An anti-pattern that fails to comply with those rules means having a multi-valued field in a database schema:

```
1  create table tweet
2   (
3     id      bigint primary key,
4     date    timestamptz,
5     message text,
6     tags    text
7   );
```

Data would then be added with a semicolon separator, for instance, or maybe a pipe | char, or in some cases with a fancy Unicode separator char such as §, ¶ or ¦. Here we find a classic semicolon:

```
         id          | date | message |          tags
═════════════════════╪══════╪═════════╪═══════════════════════
 720553530088669185  | ...  | ...     | #NY17
 720553531665682434  | ...  | ...     | #Endomondo;#endorphins
(2 rows)
```

Using PostgreSQL makes it possible to use the *regexp_split_to_array()* and *regexp_split_to_table()* functions we saw earlier, and then to process the data in a relatively sane way. The problem with going against *1NF* is that it's nearly impossible to maintain the data set as the model offers all the database anomalies listed previously.

Several things are very hard to do when you have several tags hidden in a *text* column using a separator:

- Tag Search

  To implement searching for a list of messages containing a single given tag, this model forces a *substring* search which is much less efficient than direct search.

  A normalized model would have a separate *tags* table and an association table in between the *tweet* and the *tags* reference table that we could name *tweet_tags*. Then search for tweets using a given tag is easy, as it's a simple join operation with a restriction that can be expressed either as a *where* clause or in the *join condition* directly.

  It is even possible to implement more complex searches of tweets containing several tags, or at least one tag in a list. Doing that on top of the *CSV* inspired anti-pattern is much more complex, if even possible at all.

  Rather than trying, we would fix the model!

- Usage Statistics per Tag

  For the same reasons that implementing search is difficult, this *CSV* model anti-pattern makes it hard to compute per-tag statistics, because the *tags* column is considered as a whole.

- Normalization of Tags

  People make typos or use different spellings for the tags, so we might want to normalize them in our database. As we keep the message unaltered in a different column, we would not lose any data doing so.

  While normalizing the tags at input time is trivial when using a tags reference table, it is now an intense computation, as it requires looping over all messages and splitting the tags each time.

This example looks a lot like a case of *premature optimization*, which per Donald Knuth is the root of all evil... in most cases. The exact quote reads:

> *Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*
>
> "Structured Programming with Goto Statements". Computing Surveys 6:4 (December 1974), pp. 261–301, ğ1.

Database modeling has a non-trivial impact on query performance and as such is part of making attempts at upping efficiency. Using a *CSV* formatted attribute rather than two additional tables looks like optimization, but actually it will make just about everything worse: debugging, maintenance, search, statistics, normalization, and other use cases.

UUIDs

The PostgreSQL data type UUID allows for 128 bits synthetic keys rather than 32 bits with *serial* or 64 bits with *bigserial*.

The *serial* family of data types is built on a *sequence* with a standard defined behavior for collision. A *sequence* is non-transactional to allow several concurrent transactions to each get their own number, and each transaction might then *commit* or fail to commit with a *rollback*. It means that sequence numbers are delivered in a monotonous way, always incrementally, and will be assigned and used without any ordering known in advance, and with holes in between delivered values.

Still, *sequences* and their usage as a default value for synthetic keys offer a guarantee against collisions.

UUIDs on the other hand rely on a way to produce random numbers in a 128 bits space that offers a strong theoretical guarantee against collision. You might have to retry producing a number, though very rarely.

UUIDs are useful in distributed computing where you can't synchronize every concurrent and distributed transaction against a common centralized *sequence*, which would then act as a *Single Point Of Failure*, or *SPOF*.

That said, neither sequences nor *UUID* provides a natural primary key for your data, as seen in the Primary Keys section.

Denormalization

When modeling a database schema for your application or business case, the very first step should always consist of a thorough *normalization* of the schema. This step takes time, and it's time well spent as it allows us to understand in depth the system being designed.

When reaching *3NF* then *Boyce-Codd Normal Form*, and even *4NF*, then the next step is naturally generating content and writing queries. Write queries that implement workflow oriented queries, often named *CRUD* for *create*, *read*, *update*, *delete* where the application mainly deals with a single record at a time. Also, write queries that implement a *reporting* workflow and have a broad view of your system, maybe for weekly marketing analysis, invoicing, user suggestions for upselling, or other activities that are interesting in your business field.

Once all of that is done, some difficulties may appear, either because the fully normalized schema is too heavy to deal with at the application level without any benefits, or because having a highly normalized schema involves performances penalties that you've measured and cannot tolerate.

Fully normalized schemas often have a high number of tables and references in between them. That means lots of *foreign key constraints* and lots of *join operations* in all your application queries. That said, PostgreSQL has been coded with the SQL standard and the normalization rules in mind and is very good at *join operations* in general. Also, PostgreSQL implements row-level locking for most of its operations, so the cost of constraints isn't a show stopper in a great many cases.

That said if some part of your application's workload makes it difficult to sustain a fully normalized schema, then it might be time to find trade-offs. The process of *denormalization* consists of relaxing the *normalization*

rules to reach an acceptable trade-off in terms of data quality and data maintenance.

As in any trade-off game, the techniques to apply depend on your goal: you might want to speed up reporting activities at the expense of data maintenance, or the other way around.

Premature Optimization

As seen in the previous section with the CSV model anti-pattern, database modeling makes it easy to fall into the trap of premature optimization. Only use denormalization techniques when you've made a strong case for needing them.

A strong case means you have benchmarked *your* application code against *your real production data* or a data set that has the same distribution and is as real as possible, and on a range of different server setups. A strong case also means that you've spent time rewriting SQL queries to have them pass your acceptance tests. A strong case means you know how much time a query is allowed to spend and how much time it's actually spending — in average, median, and 95 and 99 percentiles.

When there's no way to speed-up your application another way, then it is time to *denormalize* the schema, i.e. make a decision to put your data quality at risk in order to be able to serve your users and business.

In short, performance is a feature. More often than not, performance isn't the most important feature for your business. After a certain threshold, poor performance is a killer, and it must be dealt with. That's when we *denormalize* a database schema, and not before.

Functional Dependency Trade-Offs

The main way to denormalize a schema consists of breaking the *functional dependency* rules and *repeat* data at different places so that you don't have to fetch it again. When done properly, breaking the *functional dependency* rule is the same thing as implementing a *cache* in your database.

How do you know it's been done properly? When done The Right Way, the application code has an integrated *cache invalidation* mechanism. In many cases, the cache invalidation is automated, either in bulk or triggered by some events.

The Computing and Caching in SQL section in this book addresses some mechanisms meant to cache data and invalidate a cache, which may be used when denormalizing.

Denormalization with PostgreSQL

When using PostgreSQL denormalization may happen by choosing to use denormalized data types rather than an external reference table.

Many other techniques are possible to use, and some of them are listed later in this chapter. While some techniques are widespread and well known in other database management systems, some of them are unique to PostgreSQL.

When implementing any of the following denormalization techniques, please keep in mind the following rules:

- Choose and document a *single source of truth* for any and all data you are managing,

  Denormalization introduces divergence, so you will have to deal with multiple copies of the same data with differences between the copies. It needs to be clear for everybody involved and every piece of code where the *truth* is handled.

- Always implement *cache invalidation* mechanisms.

  In those times when you absolutely need to *reset* your cache and distribute the known correct version of your data, it should be as simple as running a well-known, documented, tested and maintained procedure.

- Check about concurrency behavior in tersm of data maintenance.

  Implementing *denormalization* means more complex data maintenance operations, which can be a source of reduced write-scalability

for most applications. The next chapter — Data Manipulation and Concurrency Control — dives into this topic.

To summarize, denormalization techniques are meant to optimize a database model. As it's impossible to optimize something you didn't measure, first normalize your model, benchmark it, and then see about optimizing.

Materialized Views

Back to the *f1db* database model, we now compute constructor and driver points per season. In the following query, we compute points for the ongoing season and the data set available at the time of this book's writing:

```
\set season 2017

  select drivers.surname as driver,
         constructors.name as constructor,
         sum(points) as points

    from results
         join races using(raceid)
         join drivers using(driverid)
         join constructors using(constructorid)

   where races.year = :season

group by grouping sets(drivers.surname, constructors.name)
  having sum(points) > 150
order by drivers.surname is not null, points desc;
```

Here's the result, which we know is wrong because the season was not over yet at the time of the computation. The *having* clause has been used only to reduce the number of lines to display in the book; in a real application we would certainly get all the results at once. Anyway, here's our result set:

```
  driver   │ constructor │ points
═══════════╪═════════════╪════════
 ¤         │ Mercedes    │    357
 ¤         │ Ferrari     │    318
 ¤         │ Red Bull    │    184
 Vettel    │ ¤           │    202
 Hamilton  │ ¤           │    188
 Bottas    │ ¤           │    169
(6 rows)
```

Now, your application might need to display that information often. Maybe the main dashboard is a summary of the points for constructors and drivers in the current season, and then you want that information to be readily available.

When some information is needed way more often than it changes, having a *cache* is a good idea. An easy way to build such a cache in PostgreSQL is to use a *materialized view*. This time, we might want to compute the results for all seasons and index per season:

```
begin;

create schema if not exists v;
create schema if not exists cache;

create view v.season_points as
  select year as season, driver, constructor, points
    from seasons
         left join lateral
         /*
          * For each season, compute points by driver and by constructor.
          * As we're not interested into points per season for everybody
          * involved, we don't add the year into the grouping sets.
          */
         (
            select drivers.surname as driver,
                   constructors.name as constructor,
                   sum(points) as points

              from results
                   join races using(raceid)
                   join drivers using(driverid)
                   join constructors using(constructorid)

             where races.year = seasons.year

          group by grouping sets(drivers.surname, constructors.name)
          order by drivers.surname is not null, points desc
         )
         as points
         on true
order by year, driver is null, points desc;

create materialized view cache.season_points as
  select * from v.season_points;

create index on cache.season_points(season);

commit;
```

We first create a classic *view* that computes the points every time it's referenced in queries and join operations and then build a *materialized view* on top of it. This makes it easy to see how much the *materialized view* has drifted from the authoritative version of the content with a simple *except* query. It also helps to disable the *cache* provided by the *materialized view* in your application: only change the name of the relation and have the same result set, only known to be current.

This cache now is to be invalidated after every race and implementing cache invalidation is as easy as running the following refresh materialized view query:

```
refresh materialized view cache.season_points;
```

The *cache.season_points* relation is locked out from even *select* activity while its content is being computed again. For very simple *materialized view* definitions it is possible to *refresh concurrently* and avoid locking out concurrent readers.

Now that we have a cache, the application query to retrieve the same result set as before is the following:

```
select driver, constructor, points
  from cache.season_points
 where season = 2017
   and points > 150;
```

History Tables and Audit Trails

Some business cases require having a full history of changes available for audit trails. What's usually done is to maintain live data into the main table, modeled with the rules we already saw, and model a specific history table convering where to maintain previous versions of the rows, or an archive.

A history table itself isn't a *denormalized* version of the main table but rather another version of the model entirely, with a different primary key to begin with.

What parts that might require *denormalization* for history tables are?

- Foreign key references to other tables won't be possible when those reference changes and you want to keep a history that, by definition,

doesn't change.

- The schema of your main table evolves and the history table shouldn't rewrite the history for rows already written.

The second point depends on your business needs. It might be possible to add new columns to both the main table and its history table when the processing done on the historical records is pretty light, i.e. mainly listing and comparing.

An alternative to classic history tables, when using PostgreSQL, takes advantage of the advanced data type *JSONB*.

```
create schema if not exists archive;

create type archive.action_t
    as enum('insert', 'update', 'delete');

create table archive.older_versions
 (
   table_name text,
   date       timestamptz default now(),
   action     archive.action_t,
   data       jsonb
 );
```

Then it's possible to fill in the archive *older_versions* table with data from another table:

```
insert into archive.older_versions(table_name, action, data)
     select 'hashtag', 'delete', row_to_json(hashtag)
       from hashtag
      where id = 720554371822432256
  returning table_name, date, action, jsonb_pretty(data) as data;
```

This returns:

```
-[ RECORD 1 ]----------------------------------------------------------------
table_name | hashtag
date       | 2017-09-12 23:04:56.100749+02
action     | delete
data       | {                                                              ↵
           |     "id": 720554371822432256,                                  ↵
           |     "date": "2016-04-14T10:08:00+02:00",                       ↵
           |     "uname": "Brand 1LIVESTEW",                                ↵
           |     "message": "#FB @ Atlanta, Georgia https://t.co/mUJdxaTbyC",↵
           |     "hashtags": [                                              ↵
           |         "#FB"                                                  ↵
           |     ],                                                         ↵
           |     "location": "(-84.3881,33.7489)"                           ↵
```

```
          | }
INSERT 0 1
```

When using the PostgreSQL extension hstore it is also possible to compute the *diff* between versions thanks to the support for the - operator on this data type.

Recording the data as *jsonb* or *hstore* in the history table allows for having a single table for a whole application. More importantly, it means that dealing with an application life cycle where the database model evolves is allowed as well as dealing with different versions of objects into the same archive.

As seen in the previous sections though, dealing with *jsonb* in PostgreSQL is quite powerful, but not as powerful as dealing with the full power of a structured data model with an advanced SQL engine. That said, often enough the application and business needs surrounding the history entries are relaxed compared to live data processing.

Validity Period as a Range

As we already covered in the rates example already, a variant of the historic table requirement is when your application even needs to process the data even after its date of validity. When doing financial analysis or accounting, it is crucial to relate an invoice in a foreign currency to the valid exchange rate at the time of the invoice rather than the most current value of the currency.

```
create table rates
 (
  currency text,
  validity daterange,
  rate     numeric,

  exclude using gist (currency with =,
                      validity with &&)
 );
```

An example of using this model follows:

```
  select currency, validity, rate
    from rates
   where currency = 'Euro'
     and validity @> date '2017-05-18';
```

And here's what the application would receive, a single line of data of course, thanks to the *exclude using* constraint:

```
 currency │        validity          │   rate
══════════╪══════════════════════════╪══════════
 Euro     │ [2017-05-18,2017-05-19) │ 1.240740
(1 row)
```

This query is kept fast thanks to the special *GiST* indexing, as we can see in the query plan:

```
\pset format wrapped
\pset columns 57

 explain
  select currency, validity, rate
    from rates
   where currency = 'Euro'
     and validity @> date '2017-05-18';
```

```
                          QUERY PLAN
═══════════════════════════════════════════════════════════
 Index Scan using rates_currency_validity_excl on rates ⋯
⋯ (cost=0.15..8.17 rows=1 width=34)
   Index Cond: ((currency = 'Euro'::text) AND (validity ⋯
⋯@> '2017-05-18'::date))
(2 rows)
```

So when you need to keep around values that are only valid for a period of time, consider using the PostgreSQL *range* data types and the *exclusion constraint* that guarantees no overlapping of values in your data set. This is a powerful technique.

Pre-Computed Values

In some cases, the application keeps computing the same derived values each time it accesses to the data. It's easy to pre-compute the value with PostgreSQL:

- As a default value for the column if the computation rules only include information available in the same tuple
- With a *before trigger* that computes the value and stores it into a column right in your table

Triggers are addressed later in this book with an example to solve this use case.

Enumerated Types

It is possible to use *ENUM* rather than a *reference* table.

When dealing with a short list of items, the normalized way to do that is to handle the *catalog* of accepted values in a dedicated table and reference this table everywhere your schema uses that *catalog* of values.

When using more than *join_collapse_limit* or *from_collapse_limit* relations in SQL queries, the PostgreSQL optimizer might be defeated... so in some schema using an *ENUM* data type rather than a reference table can be beneficial.

Multiple Values per Attribute

In the CSV anti-pattern database model, we saw all the disadvantages of using multiple values per attribute in general, with a text-based schema and a *separator* used in the attribute values.

Managing several values per attribute, in the same row, can help reduce how many rows your application must manage. The normalized alternative has a side table for the entries, with a reference to the main table's primary key.

Given PostgreSQL array support for searching and indexing, it is more efficient at times to manage the list of entries as an array attribute in our main table. This is particularly effective when the application often has to *delete* entries and all referenced data.

In some cases, multiple attributes each containing multiple values are needed. PostgreSQL arrays of composite type instances might then be considered. Cases when that model beats the normalized schema are rare, though, and managing this complexity isn't free.

The Spare Matrix Model

In cases where your application manages lots of optional attributes per row, most of them never being used, they can be denormalized to a JSONB extra column with those attributes, all managed into a single document.

When restricting this extra *jonsb* attribute to values never referenced anywhere else in the schema, and when the application only needs this extra data as a whole, then *jsonb* is a very good trade-off for a normalized schema.

Partitioning

Partitioning refers to splitting a table with too many rows into a set of tables each containing a part of those rows. Several kinds of partitioning are available, such as *list* or *range* partitioning. Starting in PostgreSQL 10, table partitioning is supported directly.

While partitioning isn't denormalization as such, the limits of the PostgreSQL implementation makes it valuable to include the technique in this section. Quoting the PostgreSQL documentation:

- There is no facility available to create the matching indexes on all partitions automatically. Indexes must be added to each partition with separate commands. This also means that there is no way to create a primary key, unique constraint, or exclusion constraint spanning all partitions; it is only possible to constrain each leaf partition individually.
- Since primary keys are not supported on partitioned tables, foreign keys referencing partitioned tables are not supported, nor are foreign key references from a partitioned table to some other table.
- Using the ON CONFLICT clause with partitioned tables will cause an error, because unique or exclusion constraints can only be created on individual partitions. There is no support for enforcing uniqueness (or an exclusion constraint) across an entire partitioning hierarchy.
- An UPDATE that causes a row to move from one partition to another fails, because the new value of the row fails to satisfy the implicit partition constraint of the original partition.
- Row triggers, if necessary, must be defined on individual partitions, not the partitioned table.

So when using *partitioning* in PostgreSQL 10, we lose the ability to reach even the first *normal form* by the lack of *covering* primary key. Then we

lose the ability to maintain a reference to the partitioned table with a *foreign key*.

Before partitioning any table in PostgreSQL, including PostgreSQL 10, as with any other denormalization technique (covered here or not), please do your homework: check that it's really not possible to sustain the application's workload with a normalized model.

Other Denormalization Tools

PostgreSQL extensions such as *hstore*, *ltree*, *intarray* or *pg_trgm* offer another set of interesting trade-offs to implement specific use cases.

For example ltree can be used to implement nested *category* catalogs and reference articles precisely in this catalog.

Denormalize wih Care

It's been mentioned already, and it is worth saying it again. Only denormalize your application's schema when you know what you're doing, and when you've double-checked that there's no other possibility for implementing your application and business cases with the required level of performance.

First, query optimization techniques — mainly rewriting until it's obvious for PostgreSQL how to best execute a query — can go a long way. Production examples of query rewrite improving durations from minutes to milliseconds are commonly achieved, in particular against queries written by ORMs or other naive toolings.

Second, denormalization is an optimization technique meant to leverage trade-offs. Allow me to quote Rob Pike again, as he establishes his first rule of programming in Notes on Programming in C as the following:

> *Rule 1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.*

The rule works as well for a database model as it does for a program. Maybe the database model is even more tricky because we only measure time spent by ran queries, usually, and not the time it takes to:

- Understand the database model
- Understand how to use the database model to solve a new business case
- Write the SQL queries necessary to the application code
- Validate data quality

So again, only put all those nice properties at risk with denormalizing the schema when there's no other choice.

Not Only SQL

PostgreSQL is a solid *ACID* relational database management system and uses the *SQL* language to process, manage and query the data. Its main purpose is to guarantee a consistent view of a business as a whole, at all times, while applications are concurrently active in read and write modes of operation.

To achieve a strong level of consistency, PostgreSQL needs the application designers to also design a solid data model, and at times to think about concurrency issues. We deal with those in the next chapter: Data Manipulation and Concurrency Control.

In recent years, big players in the industry faced a new scale of business, never before seen. Nowadays, a big player may have tens or hundreds of millions of concurrent users. Each user produces new data, and some business models need to react quickly to the newly inserted data and make it available to customers — mostly advertising networks...

Solving that scale of an activity introduced new challenges and the necessity to work in a *distributed* fashion. A single instance would never be able to address hundreds of millions of concurrent users, all actively producing data.

In order to be able to address such a scale, new systems have been designed that relax one or several of the *ACID* guarantees. Those systems are

grouped under the *NoSQL* flagship term and are very diverse in their capabilities and behavior. Under the *NoSQL* term, we find solutions with characteristics including:

- No support for transactions
- Lacking *atomic* operations, for which transactions are needed
- Lacking *isolation*, which means no support for *online backups*
- No query language, instead using an API
- No consistency rules, not even data types
- A reduced set of operations, often only *key/value* support
- Lacking support for *join* or *analytics* operations
- Lacking support for business constraints
- No support for *durability*

Relaxing the very strong guarantees offered by traditional database systems allows some of the *NoSQL* solution to handle more concurrent activity, often using distributed nodes of computing with a distributed data set: each node only has access to a partial set of the data.

Some of those systems then added a query language, with similarities to the well-known and established *SQL*. The *NoSQL* movement has inspired a *NewSQL* movement.

PostgreSQL offers several ways to relax its *ACID* guarantees and it can be compared favorably to most of the *NoSQL* and *NewSQL* offerings, at least until the concurrency levels can't be sustained by a single instance.

Solutions to *scale-out* PostgreSQL are readily available, either as *extensions* or as *forks*, and these are not covered by this book. In this chapter, we focus on using PostgreSQL as a *NoSQL* solution with batteries included, for those cases when you need them, such as reporting, analytics, data consistency and quality, and other business needs.

Schemaless Design in PostgreSQL

An area where the *NoSQL* systems have been prominent is in breaking with the normalization rules and the hard step of modeling a database schema. Instead, most *NoSQL* system will happily manage any data the application sends through. This is called the *schemaless* approach.

In truth, there's no such thing as a *schemaless* design actually. What it

means is that the name and type of the document properties, or fields, are hard-coded into the application code.

A readily available JSON data set is provided at https://mtgjson.com that *Provides Magic: the Gathering* card data in JSON format, using the CCo license. We can load it easily given this table definition:

```
begin;

create schema if not exists magic;

create table magic.allsets(data jsonb);

commit;
```

Then we use a small Python script:

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

import psycopg2

PGCONNSTRING = "user=appdev dbname=appdev"

if __name__ == '__main__':
    pgconn = psycopg2.connect(PGCONNSTRING)
    curs = pgconn.cursor()

    allset = open('MagicAllSets.json').read()
    allset = allset.replace("'", "''")
    sql = "insert into magic.allsets(data) values('%s')" % allset

    curs.execute(sql)
    pgconn.commit()
    pgconn.close()
```

Now, the giant *JSON* document in a single table isn't representative of the kind of *schemaless* design addressed in this chapter. It goes a little too far to push a 27 MB document containing collections of cards into a single table. We can fix this easily, though, given that we're using PostgreSQL:

```
begin;

drop table if exists magic.sets, magic.cards;

create table magic.sets
    as
select key as name, value - 'cards' as data
  from magic.allsets, jsonb_each(data);

create table magic.cards
```

```
    as
  with collection as
  (
     select key as set,
            value->'cards' as data
       from magic.allsets,
            lateral jsonb_each(data)
  )
  select set, jsonb_array_elements(data) as data
    from collection;

commit;
```

Here's how to query such a table and get data you are interested into. Note that we use the generic *contains* operator, spelled `@>`, which finds a JSON document inside another JSON document. Our *GIN* index definition above has support for exactly this operator.

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

And we get the following card, which has been found using a *GIN* index lookup over our collection of 34207 cards, in about 1.5ms on my laptop:

```
                                jsonb_pretty
═══════════════════════════════════════════════════════════════════════════
 {                                                                        ↵
     "id": "34b67f8cf8651964995bfec268498082710d4c6a",                    ↵
     "cmc": 5,                                                            ↵
     "name": "Angelic Chorus",                                            ↵
     "text": "Whenever a creature enters the battlefield under your c···
···ontrol, you gain life equal to its toughness.",                        ↵
     "type": "Enchantment",                                               ↵
     "types": [                                                           ↵
         "Enchantment"                                                    ↵
     ],                                                                   ↵
     "artist": "Jim Murray",                                              ↵
     "colors": [                                                          ↵
         "White"                                                          ↵
     ],                                                                   ↵
     "flavor": "The harmony of the glorious is a dirge to the wicked.···
···",                                                                     ↵
     "layout": "normal",                                                  ↵
     "number": "4",                                                       ↵
     "rarity": "Rare",                                                    ↵
     "manaCost": "{3}{W}{W}",                                             ↵
     "imageName": "angelic chorus",                                       ↵
     "mciNumber": "4",                                                    ↵
     "multiverseid": 129710,                                              ↵
     "colorIdentity": [                                                   ↵
```

```
        "W"                                                    ↵
    ]                                                          ↵
}
(1 row)
```

The thing with this *schemaless* design is that documents still have a structure, with fields and data types. It's just opaque to the database system and maintained in the application's code anyway.

Of course, *schemaless* means that you reach none of the *normal forms*, which have been designed as a helper to guarantee data quality in the long term.

So while PostgreSQL allows handling *schemaless* data thanks to its support for the JSON, XML, *arrays* and *composite* data types, only use this approach when you have zero data quality requirements.

Durability Trade-Offs

Durability is the *D* of the *ACID* guarantees, and it refers to the property that your database management system is not allowed to miss any *committed* transaction after a restart or a crash... any crash. It's a very strong guarantee, and it can impact performances behavior a lot.

Of course, by default, PostgreSQL applies a strong durability guarantee to every transaction. As you can read in the documentation about asynchronous commit, it's possible to relax that guarantee for enhanced write capacity.

PostgreSQL allows *synchronous_commit* to be set differently for each concurrent transaction of the system, and to be changed in-flight within a transaction. After all, this setting controls the behavior of the server at transaction commit time.

Reducing the write guarantees is helpful for sustaining some really heavy write workloads, and that's easy to do with PostgreSQL. One way to implement different *durability* policies in the same application would be to assign a different level of guarantee to different users:

```
1 create role dbowner with login;
2 create role app with login;
3
4 create role critical  with login in role app inherit;
```

```
create role notsomuch with login in role app inherit;
create role dontcare  with login in role app inherit;

alter user critical  set synchronous_commit to remote_apply;
alter user notsomuch set synchronous_commit to local;
alter user dontcare  set synchronous_commit to off;
```

Use the *dbowner* role for handling your database model and all your *DDL* scripts, and create your database with this role as the owner of it. Give enough privileges to the *app* role so that your application can use it to implement all the necessary workflows. Then the *critical*, *notsomuch* and *dontcare* roles will have the same set of privileges as the *app* role, and maybe host a different set of settings.

Now your application can pick the right connection string or user and obtain a stronger guarantee for any data changes made, with the *critical* user, or no durability guarantee with the *dontcare* user.

If you need to change the *synchronous_commit* setting in-flight, your application can use the SET LOCAL command.

It's also possible to implement such a policy entirely in the database side of things thanks to the following example trigger:

```
SET demo.threshold TO 1000;
CREATE OR REPLACE FUNCTION public.syncrep_important_delta()
  RETURNS TRIGGER
  LANGUAGE PLpgSQL
AS
$$ DECLARE
  threshold integer := current_setting('demo.threshold')::int;
  delta integer := NEW.abalance - OLD.abalance;
BEGIN
  IF delta > threshold
  THEN
    SET LOCAL synchronous_commit TO on;
  END IF;
  RETURN NEW;
END;
$$;
```

Such a trigger would have a look at the delta from your balance at commit time and depending on the amount would upgrade your *synchronous_commit* setting.

Sometimes though, even with relaxing the *durability* guarantees, business requirements can't be met with a single server handling all the write traffic.

Then, it is time to *scale out*.

Scaling Out

A very interesting area in which the *NoSQL* solutions made progress is in the ability to natively scale-out a production setup, without extra efforts. Thanks to their design choice of a reduced set of operations supported — in particular the lack of *join operations* — and a relaxed consistency requirement set — such as the lack of transaction support and the lack of integrity constraints — the *NoSQL* systems have been able to be innovative in terms of distributed computing.

Native *scale out* is achieved when it's easy to add *computing nodes* or *servers* into a production setup, at run-time, and then improve both the read and write capacity of the whole production setup.

High availability and load balancing are something separate from scale out, and can be done both by the NoSQL systems and by PostgreSQL based architectures, as covered in the PostgreSQL documentation entitled High Availability, Load Balancing, and Replication.

PostgreSQL native scale-out does not exist yet. Commercial and open-source — both at the same time — extensions and forks are available that solve this problem such as Postgres-BDR from ***2ndQuadrant*** or Citus from ***citusdata***.

PostgreSQL 10 ships with logical replication support included, and this allows for a certain level of scaling-out solutions.

If your business manages data from separated areas, say geographically independent units, then it's possible to have each geographical unit served by a separate PostgreSQL server. Then use *logical replication* to combine the data set into a single global server for a classic setup, or to local copies in each region you operate into.

The application still needs to know where is the data is that it needs to access to, so the solution isn't transparent yet. That said, in many business cases write latency is a bigger problem than write scalibility, so a federated central server is still possible to maintain, and now the reporting applications can use that PostgreSQL instance.

When considering a *scaling out* solution, always first consider the question of *online backups*: do you still need them, and if so, are they possible to implement? Most of the native scale-out systems offer no global transactions, which means no isolation from concurrent activity and as a result there is no possibility to implement a consistent online backup.

An interview with Álvaro Hernández Tortosa

IT entrepreneur, founder of two software development companies (8Kdata, Wizzbill). Software architect and developer. Open source consultant and supporter. **Álvaro Hernández Tortosa** leads the ToroDB project, a MongoDB replica solution based on PostgreSQL!

In particular, check out the Stampede product, which brings MongoDB to PostgreSQL. Stampede automagically finds the schema of your MongoDB data and presents it as relational tables and columns. Stampede is just a hidden secondary node of your MongoDB replica set. No need to design any DDL. Plug&Play!

From your experience building the ToroDB bridge in between relational and "schemaless" worlds, do you still see any advantage in the relational data model with business constraints?

> *I absolutely do. Let me really quantify it, in two very clear scenarios.*
>
> *One is my own experience with dynamic schema (please let me avoid the schema-less term, which I think it is completely flawed. I prefer dynamic schema or schema-attached). Since ToroDB replicates data that previously exists on a MongoDB instance, we needed to find applications that created data on MongoDB. Or write them. We did, of course, both. And on writing applications for MongoDB, we experience the dynamic schema on MongoDB for ourselves.*
>
> *It looks appealing at first. I can throw anything at it, and it works. You don't need to design the schema! So you start prototyping very quickly. But data almost always has "relations". For instance, we set-up an IoT device with an antenna*

(yeah, on our office's roof) to receive live flight data (ADSB receiver). And we stored the flight data in a collection. But soon you download a "database" of carriers, and you want to relate them to the flight data. And then airports. And then plane models. And then.... and how do you store all that data in MongoDB? In different collections? Embedded into the flight data documents? Otherwise? These questions typically come up very early, and pose schema design considerations. Wasn't "schema-less" something that avoided you designing the schema? Not at all. Indeed, even MongoDB recommends designing the schema as a best-practice, and they even offer full courses on schema design on MongoDB! And once you understand this and need to design the schema, you realize you are basically limited to the following options:

a. *Embed 1:1 relationships inside the documents (this is fine).*

b. *Embed 1:N relationships (de-normalization: may lead to data duplication)*

c. *Simulate N:M relationships either by embedding (you choose only one side of the* join*, forget about the other, and also leads to data duplication) or you embed* ids*, and you do the join at the application level (reinvent the wheel).*

So in any case you need to carefully design the data structure and your options are much more limited than in the relational world. That's not saying there are use cases for dynamic schema, like very flat data structures, or as a temporary store for data of very dynamic properties, which you may normalize later. But it's not the unicorn we have been told to believe it is.

The second scenario is related to analytics performance. Basically, NoSQL is not *designed for analytics purposes and perform very poorly. We found 1-2 orders of magnitude speedup when performing the same queries on relational-structured data vs. NoSQL.*

This may sound counterintuitive: after all NoSQL is for "Big Data", isn't it? Well, it could also be explained in a very in-

tuitive manner: NoSQL data is unstructured data. And unstructured data, as it name implies, is unstructured, that is, doesn't have an a priori structure. It is a bit "chaotic", unorganized. Data may be present, absent, or located anywhere. And what is analytics? Obtaining valuable information from data. But if data is unstructured, every analytic query needs to parse and analyze every single document present and infer its structure, check if the query predicate matches the document (or even if the keys that are looking for even exist on this document!) and so forth. This represents a significant extra effort that is completely not required in relational datastores. And hence they are able to perform even orders of magnitude faster. Queries that take hours in NoSQL may take just a few seconds in relational. It's this dramatic. For more information, feel free to read our blog post and benchmarks on this topic: https://www.8kdata.com/blog/announcing-torodb-stampede-1-0-beta/.

With ToroDB Stampede it's now possible to have both MongoDB and PostgreSQL feature sets on top of the same live data set. How would you compare the query languages for MongoDB and PostgreSQL?

MongoDB query language has been growing, adding new operators and functionality and I expect this trend to continue with every release. However, if you compare it feature-wise with SQL, especially with PostgreSQL's very rich feature SQL implementation, it is almost night and day. To name a couple of examples, joins are only limited in a very limited fashion, and there are no window functions. What is worse is that some query patterns in MongoDB are not optimized and performance varies dramatically from feature to feature. I expect MongoDB query language to take a long time to catch up, if that's possible, with PostgreSQL's SQL language.

Syntax is another issue. MongoDB's query language is a JSON document, and it soon becomes awkward to understand and follow. Take a moderately complex query in MongoDB and its equivalent in SQL and present them to the average developer, not specially trained in either. You will see the difference.

But the main problem I see in MongoDB, regarding its user-facing language it's the compatibility. SQL is a standard, and even if there are some minor differences between implementations and the standard itself (by the way, PostgreSQL here does a very good job, following the standard very closely), it has led to the development, for many years, of a huge ecosystem of tools and applications that you can use with the database. There is simply no such ecosystem for MongoDB, it's just a minor fraction in comparison.

Note: sure, MongoDB has the proprietary BI Connector, which theoretically allows you to connect MongoDB to any SQL tool. The true story is that BI Connector performance is very poor when compared to a SQL database, and its SQL compatibility support is also very small. So it just works on some limited number of cases.

How would you compare a pure JSON "schemaless" database such as MongoDB against PostgreSQL denormalization options such as arrays, composite types, JSONB embedded documents, etc?

PostgreSQL data types are really rich and flexible. You can very easily create your own or extend others. JSONB, in particular, emulates a whole document, and also supports quite advanced indexing (B-tree indexes on an expression of the JSON document, or specialized json indexes that index either the whole document or paths within it). One very obvious question is whether jsonb data type can compete with MongoDB on its own field, dynamic schema.

One the one hand, MongoDB is not only chosen because of the dynamic schema, but other capabilities such as built-in high-availability (with its own gotchas, but after all integrated into core) and distributed query. On the former, PostgreSQL cannot compete directly, there is no HA solution in-core, even though there are several external solutions. As for distributed queries, more related to the topic being discussed, there is also not support per se in PostgreSQL (however you may use Citus Data's PostgreSQL extension for a distributed data store or Greenplum for data warehousing capabilities). But in combination,

we cannot clearly say that PostgreSQL here offers a complete alternative to MongoDB.

On the other hand, if we're just talking about data and not database infrastructure, JSONB pretty much fulfills the purposes of a document store, and it's probably better in some areas. Probably the query language (JSONB's query functions and operators, that go beyond SQL) are less advanced than MongoDB's query language (even with all the internal issues that MongoDB query language has). But it offers the best of both worlds: you can freely combine unstructured with structured data. And this is, indeed, a very compelling use-case: the benefits of a normalized, relational schema design for the core parts of the data, and those that are obvious and clear from the beginning; and add as needed jsonb columns for less structured, more dynamic, changing data, until you can understand its shape and finally migrate to a relational schema. This is really the best of both worlds and my best recommendation.

7

Data Manipulation and Concurrency Control

Contents

In the previous chapters, we saw different ways to fetch exactly the data you're interested into from the database server. This data that we've been querying using SQL must get there, and that's the role of the *DML* parts of the standard: *data manipulation language*.

The most important aspects of this language for maintaining data are its concurrency properties with the *ACID* guarantees, and its capability to process batches of rows at a time.

The *CRUD* capabilities are essential to any application: create, read, update and delete one entry at a time is at the foundation of our applications, or at least their admin panels.

Another Small Application

In a previous chapter when introducing arrays we used a dataset of 200,000 USA geolocated tweets with a very simple data model. The data model is a direct *port* of the Excel sheet format, allowing a straightforward loading process: we used the `\copy` command from *psql*.

```
begin;

create table tweet
 (
   id          bigint primary key,
   date        date,
   hour        time,
   uname       text,
   nickname    text,
   bio         text,
   message     text,
   favs        bigint,
   rts         bigint,
   latitude    double precision,
   longitude   double precision,
   country     text,
   place       text,
   picture     text,
   followers   bigint,
   following   bigint,
   listed      bigint,
   lang        text,
   url         text
 );

\copy tweet from 'tweets.csv' with csv header delimiter ';'

commit;
```

This database model is all wrong per the *normal forms* introduced earlier:

- There's neither a *unique* constraint nor *primary key*, so there is nothing preventing insertion of duplicates entries, violating *1NF*.
- Some non-key attributes are not dependent on the key because we

mix data from the Twitter account posting the message and the message itself, violating *2NF*.

This is the case with all the user's attributes, such as the *nickname*, *bio*, *picture*, *followers*, *following*, and *listed* attributes.

- We have transitive dependencies in the model, which violates *3NF* this time.
  - The *country* and *place* attributes depend on the *location* attribute and as such should be on a separate table, such as the *geonames* data as used in the Denormalized Data Types chapter.
  - The *hour* attributes depend on the *date* attribute, as the *hour* alone can't represent when the tweet was transmitted.
- The *longitude* and *latitude* should really be a single *location* column, given PostgreSQL's ability to deal with geometric data types, here a *point*.

It is interesting to note that failing to respect the normal forms has a negative impact on application's performance. Here, each time a user changes his or her *bio*, we will have to go edit the user's *bio* in every tweet ever posted. Or we could decide to only give new tweets the new *bio*, but then at query time when showing an old tweet, it gets costly to fetch the current bio from the user.

From a concurrency standpoint, a normalized schema helps to avoid concurrent *update* activity on the same rows from occuring often in production.

It's now time to rewrite our schema, and here's a first step:

```
1  begin;
2
3  create schema if not exists tweet;
4
5  create table tweet.users
6   (
7     userid     bigserial primary key,
8     uname      text not null,
9     nickname   text not null,
10    bio        text,
11    picture    text,
12    followers  bigint,
13    following  bigint,
14    listed     bigint,
```

```
15
16    unique(name)
17  );
18
19  create table tweet.message
20  (
21    id          bigint primary key,
22    userid      bigint references tweet.users(userid),
23    datetime    timestamptz not null,
24    message     text,
25    favs        bigint,
26    rts         bigint,
27    location    point,
28    lang        text,
29    url         text
30  );
31
32  commit;
```

This model cleanly separates users and their messages and removes the attributes *country* and *place*, which we maintain separately in the geonames schema, as seen earlier.

That said, *followers* and *following* and *listed* fields are a summary of other information that we should have but don't. The fact that the extract we worked with had a simpler statistics oriented schema shouldn't blind us here. There's a better way to register relationships between users in terms of who follows who and who lists who, as in the following model:

```
1   begin;
2
3   create schema if not exists tweet;
4
5   create table tweet.users
6   (
7     userid      bigserial primary key,
8     uname       text not null,
9     nickname    text,
10    bio         text,
11    picture     text,
12
13    unique(uname)
14  );
15
16  create table tweet.follower
17  (
18    follower    bigint not null references tweet.users(userid),
19    following   bigint not null references tweet.users(userid),
20
21    primary key(follower, following)
```

```
 );

create table tweet.list
 (
   listid     bigserial primary key,
   owner      bigint not null references tweet.users(userid),
   name       text not null,

   unique(owner, name)
 );

create table tweet.membership
 (
   listid     bigint not null references tweet.list(listid),
   member     bigint not null references tweet.users(userid),
   datetime   timestamptz not null,

   primary key(listid, member)
 );

create table tweet.message
 (
   messageid  bigserial primary key,
   userid     bigint not null references tweet.users(userid),
   datetime   timestamptz not null default now(),
   message    text not null,
   favs       bigint,
   rts        bigint,
   location   point,
   lang       text,
   url        text
 );

commit;
```

Now we can begin to work with this model.

Insert, Update, Delete

The three commands insert, update, and delete have something in common: they accept a *returning* clause. This allows the *DML* command to return a result set to the application with the same protocol as the *select* clause, both are a *projection*.

This is a PostgreSQL addition to the SQL standard and it comes with clean and general semantics. Also, it avoids a network roundtrip when

your application needs to know which default value has been chosen for its own bookkeeping.

Another thing the three commands have in common is a way to do *joins*. It is spelled differently in each statement though, and it is included in the SQL standard too.

Insert Into

Given our model of tweets, the first thing we need are users. Here's how to create our first users:

```
insert into tweet.users (userid, uname, nickname, bio)
     values (default, 'Theseus', 'Duke Theseus', 'Duke of Athens.');
```

The SQL standard *values* clause is usable anywhere *select* is expected, as we saw already in our truth tables earlier. Also, values accepts several rows at a time.

```
insert into tweet.users (uname, bio)
     values ('Egeus', 'father to #Hermia.'),
            ('Lysander', 'in love with #Hermia.'),
            ('Demetrius', 'in love with #Hermia.'),
            ('Philostrate', 'master of the revels to Theseus.'),
            ('Peter Quince', 'a carpenter.'),
            ('Snug', 'a joiner.'),
            ('Nick Bottom', 'a weaver.'),
            ('Francis Flute', 'a bellows-mender.'),
            ('Tom Snout', 'a tinker.'),
            ('Robin Starveling', 'a tailor.'),
            ('Hippolyta', 'queen of the Amazons, betrothed to Theseus.'),
            ('Hermia', 'daughter to Egeus, in love with Lysander.'),
            ('Helena', 'in love with Demetrius.'),
            ('Oberon', 'king of the fairies.'),
            ('Titania', 'queen of the fairies.'),
            ('Puck', 'or Robin Goodfellow.'),
            ('Peaseblossom', 'Team #Fairies'),
            ('Cobweb', 'Team #Fairies'),
            ('Moth', 'Team #Fairies'),
            ('Mustardseed', 'Team #Fairies'),
            ('All', 'Everyone speaking at the same time'),
            ('Fairy', 'One of them #Fairies'),
            ('Prologue', 'a play within a play'),
            ('Wall', 'a play within a play'),
            ('Pyramus', 'a play within a play'),
            ('Thisbe', 'a play within a play'),
            ('Lion', 'a play within a play'),
            ('Moonshine', 'a play within a play');
```

If you have lots of rows to insert into your database, consider using the copy command instead of doing a series of *inserts*. If for some reason you can't use *copy*, for performance reasons, consider using a single transaction doing several *insert* statements each with many *values*.

Insert Into ... Select

The *insert* statement can also use a query as a data source. We could, for instance, fill in our *tweet.follower* table with people that are known to love each other from their *bio* field; and also we should have the fairies follow their queen and king, maybe.

First, we need to take this data apart from the previously inserted fields, which is our data source here.

```
select users.userid as follower,
       users.uname,
       f.userid as following,
       f.uname
  from      tweet.users
       join tweet.users f
         on f.uname = substring(users.bio from 'in love with #?(.*).')
 where users.bio ~ 'in love with';
```

The *substring* expression here returns only the regular expression matching group, which happens to be the name of who our user loves. The query then gives us the following result, which looks about right:

```
 follower |   uname   | following |   uname
══════════╪═══════════╪═══════════╪═══════════
        3 | Lysander  |        13 | Hermia
        4 | Demetrius |        13 | Hermia
       13 | Hermia    |         3 | Lysander
       14 | Helena    |         4 | Demetrius
(4 rows)
```

Now, we want to insert the *follower* and *following* data into the *tweet.follower* table of course. As the *insert into* command knows how to read its input from the result of a *select* statement, it's pretty easy to do:

```
insert into tweet.follower
     select users.userid as follower,
            f.userid as following
       from      tweet.users
            join tweet.users f
```

```
               on f.uname = substring(users.bio from 'in love with #?(.*).')
     where users.bio ~ 'in love with';
```

Now about those fairies following their queen and king:

```
with fairies as
(
  select userid
    from tweet.users
   where bio ~ '#Fairies'
)
insert into tweet.follower(follower, following)
     select fairies.userid as follower,
            users.userid as following
       from fairies cross join tweet.users
      where users.bio ~ 'of the fairies';
```

This time we even have the opportunity to use a *cross join* as we want to produce all the different combinations of a *fairy* with their royal subjects.

Here's what we have set-up in terms of followers now:

```
select follower.uname as follower,
       follower.bio as "follower's bio",
       following.uname as following

  from tweet.follower as follows

       join tweet.users as follower
         on follows.follower = follower.userid

       join tweet.users as following
         on follows.following = following.userid;
```

And here's what we've setup:

```
   follower   |              follower's bio               | following
--------------+--------------------------------------------+-----------
 Hermia       | daughter to Egeus, in love with Lysander.  | Lysander
 Helena       | in love with Demetrius.                    | Demetrius
 Demetrius    | in love with #Hermia.                      | Hermia
 Lysander     | in love with #Hermia.                      | Hermia
 Peaseblossom | Team #Fairies                              | Oberon
 Cobweb       | Team #Fairies                              | Oberon
 Moth         | Team #Fairies                              | Oberon
 Mustardseed  | Team #Fairies                              | Oberon
 Peaseblossom | Team #Fairies                              | Titania
 Cobweb       | Team #Fairies                              | Titania
 Moth         | Team #Fairies                              | Titania
 Mustardseed  | Team #Fairies                              | Titania
(12 rows)
```

The support for *select* as a source of data for the *insert* statement is the way to implement *joins* for this command.

The *insert into* clause also accepts a conflict resolution clause with the *on conflict* syntax, which is very powerful, and that we address in the isolation and locking part of this chapter.

Update

The SQL *update* statement is used to replace existing values in the database. Its most important aspect lies in its concurrency behavior, as it allows replacing existing values while other users are concurrently working with the database.

In PostgreSQL, all the concurrency feature are based on MVCC, and in the case of the *update* statement it means that internally PostgreSQL is doing both an *insert* of the new data and a *delete* of the old one. PostgreSQL system columns *xmin* and *xmax* allow visibility tracking of the rows so that concurrent statement have a consistent snapshot of the server's data set at all times.

As row locking is done per-tuple in PostgreSQL, an *update* statement only ever blocks another *update*, *delete* or *select for update* statement that targets the same row(s).

We created some users without a *nickname* before, and maybe it's time to remedy that, by assigning them their *uname* as a *nickname* for now.

```
1  begin;
2
3     update tweet.users
4        set nickname = 'Robin Goodfellow'
5      where userid = 17 and uname = 'Puck'
6  returning users.*;
7
8  commit;
```

Here we pick the id 17 from the table after a manual lookup. The idea is to show how to update fields in a single tuple from a *primary key* lookup. In a lot of cases, our application's code has fetched the *id* previously and injects it in the update query in much the same way as this.

And thanks to the *returning* clause, we get to see what we've done:

```
 userid | uname |     nickname      |         bio          | picture
════════╪═══════╪═══════════════════╪══════════════════════╪═════════
     17 | Puck  | Robin Goodfellow | or Robin Goodfellow. | ¤
(1 row)
```

As you can see in the previous query not only we used the *primary key* field, but as it is a synthetic key, we also added the real value we are interested into. Should we have pasted the information wrong, the *update* would find no matching rows and affect zero tuples.

Now there's another use case for that double check: concurrency. We know that the *Robin Goodfellow* nickname applies to *Puck*. What if someone did *update* the *uname* of *Puck* while we were running our update statement? With that double check, we know exactly one of the following is true:

- Either the other statement came in first and the name is no longear *Puck* and we updated no rows.
- The other statement will come later and we did update a row that we know is userid 17 and named *Puck*.

Think about that trick when dealing with concurrency in your application's code, and even more when you're fixing up some data from the console for a one-off fix. Then always use an explicit transaction block so that you can check what happened and issue a *rollback;* when it's not what you thought.

We can also *update* several rows at the same time. Say we want to add a default nickname to all those characters:

```
   update tweet.users
      set nickname = case when uname ~ ' '
                          then substring(uname from '[^ ]* (.*)')
                          else uname
                     end
    where nickname is null
returning users.*;
```

And now everyone is assigned a proper nickname, computed from their username with the easy and practical trick you can see in the query. The main thing to remember in that query is that you can use existing data in your *UPDATE* statement.

Now, who are our Twitter users?

```
  select uname, nickname, bio
    from tweet.users
order by userid;
```

It's a bunch of folks you might have heard about before. I've taken the names and biographies from the A Midsummer Night's Dream play from Shakespeare, for which there's a full XML transcript available at Shakespeare 2.00 thanks to ***Jon Bosak***.

uname	nickname	bio
Theseus	Duke Theseus	Duke of Athens.
Egeus	Egeus	father to #Hermia.
Lysander	Lysander	in love with #Hermia.
Demetrius	Demetrius	in love with #Hermia.
Philostrate	Philostrate	master of the revels to Theseus.
Peter Quince	Quince	a carpenter.
Snug	Snug	a joiner.
Nick Bottom	Bottom	a weaver.
Francis Flute	Flute	a bellows-mender.
Tom Snout	Snout	a tinker.
Robin Starveling	Starveling	a tailor.
Hippolyta	Hippolyta	queen of the Amazons, betrothed to Theseus.
Hermia	Hermia	daughter to Egeus, in love with Lysander.
Helena	Helena	in love with Demetrius.
Oberon	Oberon	king of the fairies.
Titania	Titania	queen of the fairies.
Puck	Robin Goodfellow	or Robin Goodfellow.
Peaseblossom	Peaseblossom	Team #Fairies
Cobweb	Cobweb	Team #Fairies
Moth	Moth	Team #Fairies
Mustardseed	Mustardseed	Team #Fairies
All	All	Everyone speaking at the same time
Fairy	Fairy	One of them #Fairies
Prologue	Prologue	a play within a play
Wall	Wall	a play within a play
Pyramus	Pyramus	a play within a play
Thisbe	Thisbe	a play within a play
Lion	Lion	a play within a play
Moonshine	Moonshine	a play within a play

(29 rows)

Inserting Some Tweets

Now that we have created a bunch of users from *A Midsummer Night's Dream*, it is time to have them tweet. The full XML transcript available at Shakespeare 2.00 contains not only the list of persona but also the full text of the play. They are all speakers and they all have lines. That's a good content for tweets!

Here's what the transcript looks like:

```
<PLAYSUBT>A MIDSUMMER NIGHT'S DREAM</PLAYSUBT>

<ACT><TITLE>ACT I</TITLE>

<SCENE><TITLE>SCENE I.  Athens. The palace of THESEUS.</TITLE>
<STAGEDIR>Enter THESEUS, HIPPOLYTA, PHILOSTRATE, and
Attendants</STAGEDIR>

<SPEECH>
<SPEAKER>THESEUS</SPEAKER>
<LINE>Now, fair Hippolyta, our nuptial hour</LINE>
<LINE>Draws on apace; four happy days bring in</LINE>
<LINE>Another moon: but, O, methinks, how slow</LINE>
<LINE>This old moon wanes! she lingers my desires,</LINE>
<LINE>Like to a step-dame or a dowager</LINE>
<LINE>Long withering out a young man revenue.</LINE>
</SPEECH>

<SPEECH>
<SPEAKER>HIPPOLYTA</SPEAKER>
<LINE>Four days will quickly steep themselves in night;</LINE>
<LINE>Four nights will quickly dream away the time;</LINE>
<LINE>And then the moon, like to a silver bow</LINE>
<LINE>New-bent in heaven, shall behold the night</LINE>
<LINE>Of our solemnities.</LINE>
</SPEECH>
```

To have the characters of the play tweet their lines, we write a simple XML parser for the format and use the *insert* SQL command. Extracted from the code used to insert the data, here's the *insert* query:

```
insert into tweet.message(userid, message)
     select userid, $2
       from tweet.users
      where users.uname = $1 or users.nickname = $1
```

As the play's text uses names such as `<SPEAKER>QUINCE</SPEAKER>` and we inserted the real name into our database, we match the play's XML content against either the *uname* or the *nickname* field.

Now that the data is loaded, we can have a look at the beginning of the play in SQL.

```
  select uname, message
    from tweet.message
         left join tweet.users using(userid)
order by messageid limit 4;
```

And yes, we can now see Shakespeare tweeting:

```
  uname    |                                message
-----------+-------------------------------------------------------------------
 Theseus   | Now, fair Hippolyta, our nuptial hour                            ↵
           | Draws on apace; four happy days bring in                         ↵
           | Another moon: but, O, methinks, how slow                         ↵
           | This old moon wanes! she lingers my desires,                     ↵
           | Like to a step-dame or a dowager                                 ↵
           | Long withering out a young man revenue.
 Hippolyta | Four days will quickly steep themselves in night;                ↵
           | Four nights will quickly dream away the time;                    ↵
           | And then the moon, like to a silver bow                          ↵
           | New-bent in heaven, shall behold the night                       ↵
           | Of our solemnities.
 Theseus   | Go, Philostrate,                                                 ↵
           | Stir up the Athenian youth to merriments;                        ↵
           | Awake the pert and nimble spirit of mirth;                       ↵
           | Turn melancholy forth to funerals;                               ↵
           | The pale companion is not for our pomp.                          ↵
           | Hippolyta, I woo'd thee with my sword,                           ↵
           | And won thy love, doing thee injuries;                           ↵
           | But I will wed thee in another key,                              ↵
           | With pomp, with triumph and with revelling.
 Egeus     | Happy be Theseus, our renowned duke!
(4 rows)
```

Delete

The *delete* statement allows marking tuples for removal. Given PostgreSQL's implementation of MVCC, it would not be wise to remove the tuple from disk at the time of the *delete*:

- First, the transaction might *rollback* and we don't know that yet.
- Second, other concurrent transactions only get to see the *delete* after *commit*, not as soon as the statement is done.

As with the *update* statement the most important part of the *delete* statement has to do with concurrency. Again, the main reason why we use a RDBMS is so that we don't have to solve the concurrency problems in our application's code, where instead we can focus on delivering an improved user experience.

The actual removal of on-disk tuples happens with *vacuum*, which the system runs in the background for you automatically thanks to its autovacuum daemon. PostgreSQL might also re-use the on-disk space for

an *insert* statement as soon as the tuple isn't visible for any transaction anymore.

Say we mistakenly added characters from another play, and we don't want to have to deal with them. First, inserting them:

```
insert into tweet.users (uname, bio)
     values ('CLAUDIUS', 'king of Denmark.'),
            ('HAMLET', 'son to the late, and nephew to the present king'),
            ('POLONIUS', 'lord chamberlain.'),
            ('HORATIO', 'friend to Hamlet'),
            ('LAERTES', 'son to Polonius'),
            ('LUCIANUS', 'nephew to the king');
```

The *delete* syntax is quite simple:

```
begin;

   delete
     from tweet.users
    where userid = 22 and uname = 'CLAUDIUS'
returning *;

commit;
```

And as usual thanks to the *returning* clause, we know exactly what we just marked for deletion:

```
 userid |  uname   | nickname |       bio        | picture
════════╪══════════╪══════════╪══════════════════╪═════════
     22 | CLAUDIUS | ¤        | king of Denmark. | ¤
(1 row)
```

Now we can also *delete* more than one row with the same command — it all depends on what we match. As the new characters inserted by mistake didn't have a part in the play we inserted our messages from, then we can use an *anti-join* to delete them based on that information:

```
begin;

with deleted_rows as
(
    delete
      from tweet.users
     where not exists
           (
             select 1
               from tweet.message
              where userid = users.userid
           )
 returning *
```

```
14 )
15 select min(userid), max(userid),
16        count(*),
17        array_agg(uname)
18   from deleted_rows;
19
20 commit;
```

And as expected we get a nice summary output of exactly what we did. This should now be your default syntax for any *delete* you have to run interactively in any database, right?

```
 min | max | count |                array_agg
═════╪═════╪═══════╪═══════════════════════════════════════
  41 |  45 |     5 | {HAMLET,POLONIUS,HORATIO,LAERTES,LUCIANUS}
(1 row)
```

It is also possible to use a *join condition* when deleting rows. It is written *using* and covered in the PostgreSQL documentation about the delete command.

Tuples and Rows

In this chapter, we've been mentioning *tuples* and *rows* at different times. There's a difference between the two: a single *row* might exist on-disk as more than one *tuple* at any time, with only one of them visible to any single transaction.

The transaction doing an *update* now sees the new version of the *row*, the new *tuple* just inserted on-disk. As long as this transaction has yet to *commit* then the rest of the world still sees the previous version of the *row*, which is another *tuple* on-disk.

While in some contexts *tuples* and *rows* are equivalent, in this chapter about DML we must be careful to use them in the right context.

Deleting All the Rows: Truncate

PostgreSQL adds to the *DML* statements the *truncate* command. Internally, it is considered to be a *DDL* rather than a *DML*. It is a very efficient way to purge a table of all of its content at once, as it doesn't follow the per-tuple MVCC system and will simply remove the data files on disk.

Note that the *truncate* command is still MVCC compliant:

```
select count(*) from foo;

begin;
truncate foo;
rollback;

select count(*) from foo;
```

Assuming there's no concurrent activity on your system when running the commands, both the counting queries naturally return the same number.

Delete but Keep a Few Rows

When cleaning up a data set, it may happen that you want to remove most of the content of a table. It could be a logs table, an audit trail that has expired or something like that. As we saw earlier when using PostgreSQL, *delete* marks the tuples as not being visible anymore and then *vacuum* does the heavy lifting in the background. It is then more efficient to create a table containing only the new rows and swap it with the old table:

```
begin;

create table new_name (like name including all);

insert into new_name
     select <column list>
       from name
      where <restrictions>;

 drop table name;
alter table new_name rename to name;

commit;
```

In the general case, as soon as you remove *most* entries from your table, this method is going to be more efficient. The trouble with that method is the level of locking required to run the *drop table* and the *alter table* statements.

Those *DDL* require an *access exclusive lock* and will block any read and write traffic to both tables while they run. If you don't have slow hours or even off-hours, then it might not be feasible for you to use this trick.

The good thing about *delete* and *vacuum* is that they can run in the middle of about any concurrent traffic of course.

Isolation and Locking

The main feature of any database system is its implementation of concurrency and full respect of the system's constraints and properties when multiple transactions are modifying the state of the system at the same time.

PostgreSQL is fully *ACID* compliant and implements *transactions isolation* so that your application's concurrency can be dealt with gracefully. Concurrency is a tricky and complex problem, and concurrency issues are often hard to reproduce. That's why it's best to rely on existing solutions for handling concurrency rather than rolling your own.

Dealing with concurrency issues in programming languages usually involves proper declaration and use of *lock*, *mutex*, and *semaphore* facilities which make a clever use of *atomic* operations as supported by your CPU, and sometimes provided by the operating system. Some programming languages such as Java offer *synchronized* blocks that in turn make use of previously listed low-level features. Other programming languages such as Erlang only implement *message passing* facilities, and handle concurrency internally (in a *mailbox* system) so that you don't have to.

SQL is a declarative programming language, where our role as developers is to declare our intention: the result we want to achieve. The implementation is then tasked with implementing our command and making it right in every detail, including concurrency behavior.

PostgreSQL implementation of the concurrency behavior is dependable and allows some user control in terms of locking aspects of your queries.

Transactions and Isolation

Given the *ACID* properties, a transaction must be *Isolated* from other concurrent transactions running in the system. It is possible to choose the level of isolation from the concurrent activity, depending on your use case.

A simple use case for isolation is *online backups*. The backup application for PostgreSQL is *pg_dump*, and the role of this application is to take a snapshot of your whole database and export it to a backup file. This requires that *pg_dump* reads are completely isolated from any concurrent write activity in the system, and this is obtained with the isolation level *repeatable read* or *serializable* as described next.

From PostgreSQL version 9.1 onward, *pg_dump* uses the isolation level *serializable*. It used to be *repeatable read* until SSI implementation... more on that later.

Transaction isolation is defined by the SQL standard and implemented in PostgreSQL:

> The SQL standard defines four levels of transaction isolation. The most strict is Serializable, which is defined by the standard in a paragraph which says that any concurrent execution of a set of Serializable transactions is guaranteed to produce the same effect as running them one at a time in some order. The other three levels are defined in terms of phenomena, resulting from interaction between concurrent transactions, which must not occur at each level. The standard notes that due to the definition of Serializable, none of these phenomena are possible at that level. (This is hardly surprising – if the effect of the transactions must be consistent with having been run one at a time, how could you see any phenomena caused by interactions?)

Still quoting the PostgreSQL documentation, here are the phenomena which are prohibited at various levels are:

- Dirty read

  A transaction reads data written by a concurrent uncommitted transaction.

- Nonrepeatable read

  A transaction re-reads data it has previously read and finds that data has been modified by another transaction (that committed since the initial read).

- Phantom read

  A transaction re-executes a query returning a set of rows that satisfy

a search condition and finds that the set of rows satisfying the condition has changed due to another recently committed transaction.

- Serialization anomaly

  The result of successfully committing a group of transactions is inconsistent with all possible orderings of running those transactions one at a time.

There are four isolation levels defined by the standard: *read uncommitted*, *read committed*, *repeatable read*, and *serializable*. PostgreSQL doesn't implement *read uncommitted*, which allows *dirty reads*, and instead defaults to *read committed*.

The definition of those isolation levels says that *read committed* disallows *dirty read* anomalies, *repeatable read* disallows *dirty read* and *nonrepeatable read*, and *serializable* disallows all anomalies.

PostgreSQL also disallows *phantom read* from *repeatable read* isolation level.

About SSI

PostgreSQL's implementation of *serializable* is an amazing work. It is described in details at the PostgreSQL wiki page entitled Serializable, and the wiki page SSI contains more details about how to use it.

It took about 20 years for the research community to come up with a satisfying mathematical model for implementing *serializable snapshot isolation* in an efficient way, and then a single year for that major progress to be included in PostgreSQL!

Concurrent Updates and Isolation

In our *tweet* model of an application, we can have a look at handling *retweets*, which is a *counter* field in the *tweet.message* table. Here's how to make a *retweet* in our model:

```
1 update tweet.message
2    set rts = rts + 1
3  where messageid = 1;
```

Now, what happens if two users are doing that at the same time?

To better understand what *at the same time* means here, we can write the query extended with manual transaction control, as PostgreSQL will do when sent a single command without an explicit transaction:

```
1  begin;
2
3     update tweet.message
4        set rts = rts + 1
5      where messageid = 1;
6  returning messageid, rts;
7
8  commit;
```

Now, rather than doing this query, we open a *psql* prompt and send in:

```
1  begin;
2
3     update tweet.message
4        set rts = rts + 1
5      where messageid = 1
6  returning messageid, rts;
```

We get the following result now:

```
 messageid | rts
═══════════╪═════
         1 |   2
(1 row)
```

The transaction remains open (it's *idle in transaction*) and waits for us to do something else, or maybe *commit* or *rollback* the transaction.

Now, open a second *psql* prompt and send in the exact same query. This time the *update* doesn't return. There's no way it could: the first transaction is not done yet and is working on the row where *messageid = 1*. Until the first transaction is done, no concurrent activity can take place on this specific row.

So we go back to the first prompt and *commit*.

Then what happens depends on the *isolation level* required. Here we have the default isolation level *read committed*, and at the second prompt the *update* command is unlocked and proceeds to immediately return:

```
 messageid | rts
═══════════╪═════
         1 |   3
(1 row)
```

Now for the following examples, we need to review our *psql* setting for the *ON_ERROR_ROLLBACK* feature. When set to *true* or *interactive*, then *psql* issues savepoints to protect each outer transaction state, and that will hide what we're showing next. Type the following command to momentarily disable this helpful setting, so that we can see what really happens:

```
\set ON_ERROR_ROLLBACK off
```

If we pick the isolation level *repeatable read*, with the following syntax:

```
start transaction isolation level repeatable read;

   update tweet.message
      set rts = rts + 1
    where messageid = 1
returning messageid, rts;
```

Again, we leave the transaction open, switch to a second prompt and do the same thing, and only then — while the second update is waiting for the first transaction to finish — commit the first transactions. What we get this time is this:

```
ERROR:  could not serialize access due to concurrent update

yesql!# commit;
ROLLBACK
```

Also notice that even if we ask for a *COMMIT*, what we get is a *ROLLBACK*. Once an error occurs in a transaction, in PostgreSQL, the transaction can't commit anymore.

When using the isolation level *serializable*, the same behavior as for *repeatable read* is observed, with exactly the same error message exactly.

Modeling for Concurrency

We should have another modeling pass on the *tweet.message* table now. With what we learned about concurrency in PostgreSQL, it's easy to see that we won't get anywhere with the current model. Remember when Donald Knuth said

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil. Yet we should not pass up our opportunities in that critical 3%.*

Database systems have been designed to handle concurrency so that your application's code doesn't have to. One part for the critical 3% is then related to concurrent operations, and the one that is impossible to implement in a both fast and correct way is a concurrent *update* on the same target row.

In our model here, given how the application works, we know that messages will get concurrent *update* activity for the *favs* and *rts* counters. So while the previous model looks correct with respect to *normal forms* — the counters are dependent on the message's key — we know that concurrent activity is going to be hard to handle in production.

So here's a smarter version of the *activity* parts of the database model:

```
begin;

create type tweet.action_t
    as enum('rt', 'fav', 'de-rt', 'de-fav');

create table tweet.activity
 (
  id          bigserial primary key,
  messageid   bigint not null references tweet.message(messageid),
  datetime    timestamptz not null default now(),
  action      tweet.action_t not null,

  unique(messageid, datetime, action)
 );

commit;
```

In this version, the counters have disappeared, replaced by a full record of the base information needed to compute them. We now have an *activity* list with a denormalized *ENUM* for possible actions.

To get the *rts* and *favs* counters back from this schema, we count lines in the *activity* records associated with a given *messageid*:

```
  select   count(*) filter(where action = 'rt')
         - count(*) filter(where action = 'de-rt')
         as rts,
           count(*) filter(where action = 'fav')
         - count(*) filter(where action = 'de-fav')
         as favs
    from tweet.activity
         join tweet.message using(messageid)
   where messageid = :id;
```

Reading the current counter value has become quite complex when compared to just adding a column to your query output list. On the other hand, when adding a *rt* or a *fav* action to a message, we transform the SQL:

```
update tweet.message set rts = rts +1 where messageid = :id;
```

This is what we use instead:

```
insert into tweet.activity(messageid, action) values(:id, 'rt');
```

The reason why replacing an *update* with an *insert* is interesting is concurrency behavior and locking. In the first version, retweeting has to wait until all concurrent retweets are done, and the business model wants to sustain as many concurrent activities on the same small set of messages as possible (read about *influencer* accounts).

The *insert* has no concurrency because it targets a row that doesn't exist yet. We register each action into its own tuple and require no locking to do that, allowing our production setup of PostgreSQL to sustain a much larger load.

Now, computing the counters each time we want to display them is costly. And the counters are displayed on every tweet message. We need a way to *cache* that information, and we'll see about that in the Computing and Caching in SQL section.

Putting Concurrency to the Test

When we *benchmark* the concurrency properties of the two statements above, we quickly realize that the *activity* table is badly designed. The unique constraint includes a *timestamptz* field, which in PostgreSQL is only precise down to the microsecond.

This kind of made-up *unique* constraint means we now have these errors to deal with:

```
Error: Database error 23505: duplicate key value violates unique      ⏎
  constraint "activity_messageid_datetime_action_key"
DETAIL: Key (messageid, datetime, action)                             ⏎
           =(2, 2017-09-19 18:00:03.831818+02, rt) already exists.
```

The best course of action here is to do this:

```
    alter table tweet.activity
drop constraint activity_messageid_datetime_action_key;
```

Now we can properly compare the concurrency scaling of the *insert* and the *update* based version. In case you might be curious about it, here's the testing code that's been used:

```
(defpackage #:concurrency
  (:use #:cl #:appdev)
  (:import-from #:lparallel
                #:*kernel*
                #:make-kernel #:make-channel
                #:submit-task #:receive-result
                #:kernel-worker-index)
  (:import-from #:cl-postgres-error
                #:database-error)
  (:export      #:*connspec*
                #:concurrency-test))

(in-package #:concurrency)

(defparameter *connspec* '("appdev" "dim" nil "localhost"))

(defparameter *insert-rt*
  "insert into tweet.activity(messageid, action) values($1, 'rt')")

(defparameter *update-rt*
  "update tweet.message set rts = coalesce(rts, 0) + 1 where messageid = $1")

(defun concurrency-test (workers retweets messageid
                         &optional (connspec *connspec*))
  (format t "Starting benchmark for updates~%")
  (with-timing (rts seconds)
      (run-workers workers retweets messageid *update-rt* connspec)
    (format t "Updating took ~f seconds, did ~d rts~%" seconds rts))

  (format t "~%")

  (format t "Starting benchmark for inserts~%")
  (with-timing (rts seconds)
      (run-workers workers retweets messageid *insert-rt* connspec)
    (format t "Inserting took ~f seconds, did ~d rts~%" seconds rts)))

(defun run-workers (workers retweets messageid sql
                    &optional (connspec *connspec*))
  (let* ((*kernel* (lparallel:make-kernel workers))
         (channel  (lparallel:make-channel)))
    (loop repeat workers
       do (lparallel:submit-task channel #'retweet-many-times
                                 retweets messageid sql connspec))

```

```
    (loop repeat workers sum (lparallel:receive-result channel))))

(defun retweet-many-times (times messageid sql
                           &optional (connspec *connspec*))
  (pomo:with-connection connspec
    (pomo:query
     (format nil "set application_name to 'worker ~a'"
             (lparallel:kernel-worker-index)))
    (loop repeat times sum (retweet messageid sql))))

(defun retweet (messageid sql)
  (handler-case
      (progn
        (pomo:query sql messageid)
        1)
    (database-error (c)
      (format t "Error: ~a~%" c)
      0)))
```

Here's a typical result with a concurrency of 100 workers all wanting to do 10 retweet in a loop using a *messageid*, here message 3. While it's not representative to have them loop 10 times to retweet the same message, it should help create the concurrency effect we want to produce, which is having several concurrent transactions waiting in turn in order to have a lock access to the same row.

The theory says that those concurrent users will have to wail in line, and thus spend time waiting for a lock on the PostgreSQL server. We should see that in the timing reports as a time difference:

```
CL-USER> (concurrency::concurrency-test 100 10 3)
Starting benchmark for updates
Updating took 3.099873 seconds, did 1000 rts

Starting benchmark for inserts
Inserting took 2.132164 seconds, did 1000 rts
```

The *update* variant of the test took almost 50% as much time to complete than the *insert* variant, with this level of concurrency. Given that we have really simple SQL statements, we can attribute the timing difference to having had to wait in line. Basically, the *update* version spent almost 1 second out of 3 seconds waiting for a free slot.

In another test with even more concurrency pressure at 50 retweets per worker, we can show that the results are repeatable:

```
CL-USER> (concurrency::concurrency-test 100 50 6)
Starting benchmark for updates
```

```
Updating took 5.070135 seconds, did 5000 rts

Starting benchmark for inserts
Inserting took 3.739505 seconds, did 5000 rts
```

If you know that your application has to scale, think about how to avoid concurrent activity that competes against a single shared resource. Here, this shared resource is the *rts* field of the *tweet.message* row that you target, and the concurrency behavior is going to be fine if the retweet activity is well distributed. As soon as many users want to retweet the same message, then the *update* solution has a non-trivial scalability impact.

Now, we're going to implement the *tweet.activity* based model. In this model, the number of *retweets* needs to be computed each time we display it, and it's part of the visible data. Also, in the general case, it's impossible for our users to know for sure how many retweets have been made so that we can implement a cache with *eventual consistency* properties.

Computing and Caching in SQL

There's a pretty common saying:

> *There are only two hard things in computer science: cache invalidation and naming things.*
>
> — Phil Karlton

More about that saying can be read at the Two Hard Things page from *Martin Fowler*, who tries to track it back to its origins.

It is time that we see about how to address the cache problems in SQL. Creating a set of values for caching is of course really easy as it usually boils down to writing a SQL query. Any SQL query executed by PostgreSQL uses a snapshot of the whole database system. To create a cache from that snapshot, the simplest way is to use the *create table as* command.

```
create table tweet.counters as
  select   count(*) filter(where action = 'rt')
         - count(*) filter(where action = 'de-rt')
         as rts,
           count(*) filter(where action = 'fav')
         - count(*) filter(where action = 'de-fav')
```

```
        as favs
    from tweet.activity
         join tweet.message using(messageid);
```

Now we have a *tweet.counters* table that we can use whenever we need the numbers of *rts* or *favs* from a tweet message. How do we update the counters? That's the cache invalidation problem quoted above, and we'll come to the answer by the end of this chapter!

Views

Views allow integrating server-side computations in the definition of a relation. The computing still happens dynamically at query time and is made transparent to the client. When using a view, there's no problem with *cache invalidation*, because nothing gets cached away.

```
create view tweet.message_with_counters
      as
  select messageid,
         message.userid,
         message.datetime,
         message.message,
           count(*) filter(where action = 'rt')
         - count(*) filter(where action = 'de-rt')
         as rts,
           count(*) filter(where action = 'fav')
         - count(*) filter(where action = 'de-fav')
         as favs,
         message.location,
         message.lang,
         message.url
    from tweet.activity
         join tweet.message using(messageid)
group by message.messageid, activity.messageid;
```

Given this view, the application code can query *tweet.message_with_counters* and process the same relation as in the first normalized version of our schema. The view hides the *complexity* of how to obtain the counters from the schema.

```
  select messageid,
         rts,
         nickname
    from tweet.message_with_counters
         join tweet.users using(userid)
   where messageid between 1 and 6
order by messageid;
```

We can see that I played with the generating some retweets in my local testing, done mainly over the six first messages:

```
 messageid │  rts   │   nickname
═══════════╪════════╪══════════════
         1 │  20844 │ Duke Theseus
         2 │ 111345 │ Hippolyta
         3 │  11000 │ Duke Theseus
         5 │   3500 │ Duke Theseus
         6 │  15000 │ Egeus
(5 rows)
```

That view now embeds the computation details and abstracts them away from the application code. It allows having several parts of the application deal with the same way of counting *retweets* and *favs*, which might come to be quite important if you have different backends for reporting, data analysis, and user analytics products that you're selling, or using it to sell advertising, maybe. It might even be that those parts are written in different programming languages, yet they all want to deal with the same numbers, a shared *truth*.

The view embeds the computation details, and still it computes the result each time it's referenced in a query.

Materialized Views

It is easy enough to cache a snapshot of the database into a permanent relation for later querying thanks to PostgreSQL implementation of *materialized views*:

```
1  create schema if not exists twcache;
2
3  create materialized view twcache.message
4      as select messageid, userid, datetime, message,
5                rts, favs,
6                location, lang, url
7           from tweet.message_with_counters;
8
9  create unique index on twcache.message(messageid);
```

As usual, read the PostgreSQL documentation about the command CREATE MATERIALIZED VIEW for complete details about the command and its options.

The application code can now query *twcache.message* instead of *tw.message*

and get the extra pre-computed columns for *rts* and *favs* counter. The information in the materialized view is static: it is only updated with a specific command. We have effectively implemented a cache in SQL, and now we have to solve the *cache invalidation* problem: as soon as a new action (retweet or favorite) happens on a message, our cache is wrong.

Now that we have created the cache, we run another benchmark with 100 workers doing each 100 retweets on *messageid* 3:

```
CL-USER> (concurrency::concurrency-test 100 100 3)
Starting benchmark for updates
Updating took 8.132917 seconds, did 10000 rts

Starting benchmark for inserts
Inserting took 6.684597 seconds, did 10000 rts
```

Then we query our cache again:

```
  select messageid,
         rts,
         nickname,
         substring(message from E'[^\n]+') as first_line
    from twcache.message
         join tweet.users using(userid)
   where messageid = 3
order by messageid;
```

We can see that the *materialized view* is indeed a cache, as it knows nothing about the last round of retweets that just happened:

```
 messageid | rts  |   nickname    |     first_line
═══════════╪══════╪═══════════════╪════════════════════
         3 | 1000 | Duke Theseus  | Go, Philostrate,
(1 row)
```

Of course, as every PostgreSQL query uses a database snapshot, the situation when the counter is already missing actions already happens with a table and a view already. If some *insert* are *committed* on the *tweet.activity* table while the *rts* and *favs* count query is running, the result of the query is not counting the new row, which didn't make it yet at the time when the query snapshot had been taken. *Materialized view* only extends the cache *time to live*, if you will, making the problem more obvious.

To invalidate the cache and compute the data again, PostgreSQL implements the refresh materialized view command:

```
refresh materialized view concurrently twcache.message;
```

This command makes it possible to implement a *cache invalidation policy*. In some cases, a business only analyses data up to the day before, in which case you can *refresh* your materialized views every night: that's your cache invalidation policy.

Once the *refresh materialized view* command has been processed, we can query the cache again. This time, we get the expected answer:

```
 messageid │  rts  │   nickname   │    first_line
═══════════╪═══════╪══════════════╪══════════════════
         3 │ 11000 │ Duke Theseus │ Go, Philostrate,
(1 row)
```

In the case of instant messaging such as Twitter, maybe the policy would require *rts* and *favs* counters to be as fresh as *five minutes ago* rather than *yesterday*. When the *refresh materialized view* command runs in less than five minutes then implementing the policy is a matter of scheduling that command to be executed every five minutes, using for example the *cron* Unix task scheduler.

Triggers

When a cache refresh policy of minutes isn't advisable, a common approach is to implement event-based processing. Most SQL systems, including PostgreSQL, implement an event-based facility called a *trigger*.

A *trigger* allows registering a procedure to be executed at a specified timing when an event is produced. The timing can be *before*, *after* or *instead of*, and the event can be *insert*, *update*, *delete* or *truncate*. As usual, the PostgreSQL documentation covers the topic in full details and is available online, in our case now at the manual page for the commandCREATE TRIGGER.

Many triggers in PostgreSQL are written in the PL/pgSQL — SQL Procedural Language, so we also need to read the PLpgSQL trigger procedures documentation for completeness.

Note that with PostgreSQL, it is possible to write procedures and triggers in other programming languages. Default PostgreSQL builds include support for PL/Tcl, PL/Perl, PL/Python and of course C-language functions.

PostgreSQL extensions for other programming languages are available too, maintained separately from the PostgreSQL core. You can find PL/Java, PL/v8 for Javascript powered by the V8 engine, or PL/XSLT as we saw in this book already. For even more programming language support, see the PL Matrix in the PostgreSQL wiki.

Unfortunately, it is not possible to write triggers in plain SQL language, so we have to write stored procedures to benefit from the PostgreSQL trigger capabilities.

Transactional Event Driven Processing

PostgreSQL triggers call a registered procedure each time one of the supported events is committed. The execution of the procedure is always taken as a part of the transaction, so if your procedure fails at runtime then the transaction is aborted.

A classic example of an event driven processing with a trigger in our context is to update the counters of *rts* and *favs* each time there's a related insert in the *tweet.activity* table.

```
1  begin;
2
3  create table twcache.daily_counters
4   (
5     day     date not null primary key,
6     rts     bigint,
7     de_rts  bigint,
8     favs    bigint,
9     de_favs bigint
10  );
11
12 create or replace function twcache.tg_update_daily_counters ()
13  returns trigger
14  language plpgsql
15 as $$
16 declare
17 begin
18       update twcache.daily_counters
19          set rts = case when NEW.action = 'rt'
20                         then rts + 1
21                         else rts
22                    end,
23              de_rts = case when NEW.action = 'de-rt'
24                         then de_rts + 1
25                         else de_rts
```

```
                    end,
             favs = case when NEW.action = 'fav'
                         then favs + 1
                         else favs
                     end,
             de_favs = case when NEW.action = 'de-fav'
                         then de_favs + 1
                         else de_favs
                     end
       where daily_counters.day = current_date;

  if NOT FOUND
  then
      insert into twcache.daily_counters(day, rts, de_rts, favs, de_favs)
           select current_date,
                  case when NEW.action = 'rt'
                       then 1 else 0
                    end,
                  case when NEW.action = 'de-rt'
                       then 1 else 0
                   end,
                  case when NEW.action = 'fav'
                       then 1 else 0
                   end,
                  case when NEW.action = 'de-fav'
                       then 1 else 0
                   end;
  end if;

  RETURN NULL;
end;
$$;

CREATE TRIGGER update_daily_counters
         AFTER INSERT
            ON tweet.activity
      FOR EACH ROW
       EXECUTE PROCEDURE twcache.tg_update_daily_counters();

insert into tweet.activity(messageid, action)
     values (7, 'rt'),
            (7, 'fav'),
            (7, 'de-fav'),
            (8, 'rt'),
            (8, 'rt'),
            (8, 'rt'),
            (8, 'de-rt'),
            (8, 'rt');

select day, rts, de_rts, favs, de_favs
  from twcache.daily_counters;

```

```
78 | rollback;
```

Again, we don't really want to have that trigger in our setup, so the transaction ends with a *ROLLBACK*. It's also a good way to try in-progress development in *psql* in an interactive fashion, and fix all the bugs and syntax errors until it all works.

Without this trick, then parts of the script pass and others fail, and you then have to copy and paste your way around until it's all okay, but then you're never sure that the whole script will pass from the start again, because the conditions in which you want to apply have been altered on the partially successful runs.

Here's the result of running our trigger test script:

```
BEGIN
CREATE TABLE
CREATE FUNCTION
CREATE TRIGGER
INSERT 0 8
    day      | rts | de_rts | favs | de_favs
═════════════╪═════╪════════╪══════╪═════════
 2017-09-21 |   5 |      1 |    1 |       1
(1 row)

ROLLBACK
```

The thing is, each time there's a *tweet.activity* inserted this trigger will transform the *insert* into an *update* against a single row, and the same target row for a whole day.

This implementation is totally killing any ambitions we might have had about concurrency and scalability properties of our model, in a single trigger. Yet it's easy to write such a trigger, so it's seen a lot in the wild.

Trigger and Counters Anti-Pattern

You might also notice that this triggers is very wrong in its behavior, as coded. The implementation of the *insert or update* — a.k.a. *upsert* — is coded in a way to leave the door open to concurrency issues. To understand those issues, we need to consider what happens when we start a new day:

1. The first transaction of the day attempts to *update* the daily counters table for this day, but finds no records because it's the first one.

2. The first transaction of the day then *inserts* the first value for the day with ones and zeroes for the counters.
3. The second transaction of the day then executes the *update* to the daily counter, finds the existing row, and skips the *insert* part of the trigger.

That's the happy scenario where no problem occurs. Now, in the real life, here's what will sometimes happen. It's not always, mind you, but not never either. Concurrency bugs — they like to hide in plain sight.

1. The first transaction of the day attempts to *update* the daily counters table for this day but finds no records because it's the first one.
2. The second transaction of the day attempts to *update* the daily counters table for this day, but finds no records, because the first one isn't there yet.
3. The second transaction of the day now proceeds to *insert* the first value for the day, because the job wasn't done yet.
4. The first transaction of the day then *inserts* the first value... and fails with a *primary key* conflict error because that *insert* has already been done. Sorry about that!

There are several ways to address this issue, and the classic one is documented at A PL/pgSQL Trigger Procedure For Maintaining A Summary Table example in the PostgreSQL documentation.

The solution there is to *loop* over attempts at *update* then *insert* until one of those works, ignoring the `UNIQUE_VIOLATION` exceptions in the process. That allows implementing a fall back when another transaction did insert a value concurrently, i.e. in the middle of the `NOT FOUND` test and the consequent *insert*.

Starting in PostgreSQL 9.5 with support for the *on conflict* clause of the *insert into* command, there's a much better way to address this problem.

Fixing the Behavior

While it's easy to maintain a *cache* in an event driven fashion thanks to PostgreSQL and its trigger support, turning an *insert* into an *update* with

contention on a single row is never a good idea. It's even a classic anti-pattern.

Here's a modern way to fix the problem with the previous trigger implementation, this time applied to a per-message counter of *retweet* and *favorite* actions:

```
begin;

create table twcache.counters
 (
   messageid  bigint not null references tweet.message(messageid),
   rts        bigint,
   favs       bigint,

   unique(messageid)
 );

create or replace function twcache.tg_update_counters ()
 returns trigger
 language plpgsql
as $$
declare
begin
   insert into twcache.counters(messageid, rts, favs)
        select NEW.messageid,
               case when NEW.action = 'rt' then 1 else 0 end,
               case when NEW.action = 'fav' then 1 else 0 end
   on conflict (messageid)
     do update
           set rts = case when NEW.action = 'rt'
                          then counters.rts + 1

                          when NEW.action = 'de-rt'
                          then counters.rts - 1

                          else counters.rts
                     end,

               favs = case when NEW.action = 'fav'
                           then counters.favs + 1

                           when NEW.action = 'de-fav'
                           then counters.favs - 1

                           else counters.favs
                      end
         where counters.messageid = NEW.messageid;

  RETURN NULL;
end;
```

```
$$;

CREATE TRIGGER update_counters
         AFTER INSERT
            ON tweet.activity
      FOR EACH ROW
       EXECUTE PROCEDURE twcache.tg_update_counters();

insert into tweet.activity(messageid, action)
     values (7, 'rt'),
            (7, 'fav'),
            (7, 'de-fav'),
            (8, 'rt'),
            (8, 'rt'),
            (8, 'rt'),
            (8, 'de-rt'),
            (8, 'rt');

select messageid, rts, favs
  from twcache.counters;

rollback;
```

And here's the result of running that file in *psql*, either from the command line with *psql -f* or with the interactive `\i <path/to/file.sql` command:

```
BEGIN
CREATE TABLE
CREATE FUNCTION
CREATE TRIGGER
INSERT 0 8
 messageid | rts | favs
═══════════╪═════╪══════
         7 │   1 │    0
         8 │   3 │    0
(2 rows)

ROLLBACK
```

You might have noticed that the file ends with a *ROLLBACK* statement. That's because we don't really want to install such a trigger, it's meant as an example only.

The reason why we don't actually want to install it is that it would cancel all our previous efforts to model for tweet activity scalability by transforming every *insert into tweet.activity* into an *update twcache.counters* on the same *messageid*. We looked into that exact thing in the previous section and we saw that it would never scale to our requirements.

Event Triggers

Event triggers are another kind of triggers that only PostgreSQL supports, and they allow one to implement triggers on any event that the source code integrates. Currently event triggers are mainly provided for DDL commands.

Have a look at "A Table Rewrite Event Trigger Example" in the PostgreSQL documentation for more information about event triggers, as they are not covered in this book.

Listen and Notify

The PostgreSQL protocol includes a streaming protocol with *COPY* and also implements asynchronous messages and notifications. This means that as soon as a connection is established with PostgreSQL, the server can send messages to the client even when the client is idle.

PostgreSQL Notifications

Messages that flow from the server to the connected client should be processed by the client. It could be that the server is being restarted, or an application message is being delivered.

Here's an example of doing this:

```
yesql# listen channel;
LISTEN

yesql# notify channel, 'foo';
NOTIFY
Asynchronous notification "channel" with payload "foo"  ⏎
received from server process with PID 40430.
```

Note that the message could be sent from another connection, so try it and see with several *psql* instances. The *payload* from the message can be any text, up to 8kB in length. This allows for rich messages to flow, such as JSON encoded values.

PostgreSQL Event Publication System

In the Triggers section we saw that in order to maintain a cache of the action counters either by day or by messageid, we can write a trigger. This implements event driven processing but kills our concurrency and scalability properties.

It's possible for our trigger to *notify* an external client. This client must be a daemon program, which uses *listen* to register our messages. Each time a notification is sent, the daemon program processes it as necessary, possibly updating our *twcache.counters* table. As we have a single daemon program listening to notifications and updating the cache, we now bypass the concurrency issues.

Before implementing the client application, we can implement the trigger for notification, and use *psql* as a testing client:

```
begin;

create or replace function twcache.tg_notify_counters ()
 returns trigger
 language plpgsql
as $$
declare
  channel text := TG_ARGV[0];
begin
  PERFORM (
     with payload(messageid, rts, favs) as
     (
       select NEW.messageid,
              coalesce(
                 case NEW.action
                   when 'rt'    then  1
                   when 'de-rt' then -1
                  end,
                 0
              ) as rts,
              coalesce(
                case NEW.action
                  when 'fav'    then  1
                  when 'de-fav' then -1
                 end,
                0
              ) as favs
     )
     select pg_notify(channel, row_to_json(payload)::text)
       from payload
  );
```

```
  RETURN NULL;
end;
$$;

CREATE TRIGGER notify_counters
         AFTER INSERT
            ON tweet.activity
      FOR EACH ROW
       EXECUTE PROCEDURE twcache.tg_notify_counters('tweet.activity');

commit;
```

Then to test the trigger, we can issue the following statements at a *psql* prompt:

```
listen "tweet.activity";

insert into tweet.activity(messageid, action)
     values (33, 'rt'),
            (33, 'rt'),
            (33, 'de-rt'),
            (33, 'fav'),
            (33, 'de-fav'),
            (33, 'rt'),
            (33, 'fav');
```

We get then the following output from the console:

```
INSERT 0 7
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":1,"favs":0}" received from              ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":-1,"favs":0}" received from             ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":0,"favs":1}" received from              ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":0,"favs":-1}" received from             ⏎
server process with PID 73216.
```

So we made seven inserts, and we have four notifications. This behavior might be surprising, yet it is fully documented on the PostgreSQL manual page for the NOTIFY command:

> If the same channel name is signaled multiple times from the same transaction with identical payload strings, the database server can decide to deliver a single notification only. On the other hand, notifications with distinct payload strings will always be delivered as distinct notifications. Similarly, notifications from different transactions will never get folded into one

> notification. Except for dropping later instances of duplicate notifications, NOTIFY guarantees that notifications from the same transaction get delivered in the order they were sent. It is also guaranteed that messages from different transactions are delivered in the order in which the transactions committed.

Our test case isn't very good, so let's write another one, and keep in mind that our implementation of the cache server with *notify* can only be correct if the main application issues only distinct *tweet.activity* actions in a single transaction. For our usage, this is not a deal-breaker, so we can fix our tests.

```
insert into tweet.activity(messageid, action) values (33, 'rt');
insert into tweet.activity(messageid, action) values (33, 'de-rt');
insert into tweet.activity(messageid, action) values (33, 'fav');
insert into tweet.activity(messageid, action) values (33, 'de-rt');
```

And this time we get the expected notifications:

```
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":1,"favs":0}" received from              ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":-1,"favs":0}" received from             ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":0,"favs":1}" received from              ⏎
server process with PID 73216.
Asynchronous notification "tweet.activity" with payload        ⏎
"{"messageid":33,"rts":-1,"favs":0}" received from             ⏎
server process with PID 73216.
```

Notifications and Cache Maintenance

Now that we have the basic server-side infrastructure in place, where PostgreSQL is the server and a backend application the client, we can look into about maintaining our *twcache.counters* cache in an event driven fashion.

PostgreSQL LISTEN and NOTIFY support is perfect for maintaining a cache. Because notifications are only delivered to client connections that are listening at the moment of the notify call, our cache maintenance service must implement the following behavior, in this exact order:

1. Connect to the PostgreSQL database we expect notifications from and issue the *listen* command.

2. Fetch the current values from their *single source of truth* and reset the cache with those computed values.

3. Process notifications as they come and update the in-memory cache, and once in a while synchronize the in-memory cache to its materialized location, as per the cache invalidation policy.

The cache service can be implemented within the cache maintenance service. As an example, a cache server application might both process notifications and serve the current cache from memory over an HTTP API. The cache service might also be one of the popular cache solutions such as Memcached or Redis.

In our example, we implement a cache maintenance service in Go and the cache itself is maintained as a PostgreSQL table:

```
1  begin;
2
3  create schema if not exists twcache;
4
5  create table twcache.counters
6   (
7     messageid   bigint not null primary key,
8     rts         bigint,
9     favs        bigint
10  );
11
12 commit;
```

With this table, implementing a NOTIFY client service that maintains the cache is easy enough to do, and here's what happens when the service runs and we do some testing:

```
2017/09/21 22:00:36 Connecting to postgres:///yesql?sslmode=disable···
2017/09/21 22:00:36 Listening to notifications on channel "tweet.activity"
2017/09/21 22:00:37 Cache initialized with 6 entries.
2017/09/21 22:00:37 Start processing notifications, waiting for events···
2017/09/21 22:00:42 Received event: {"messageid":33,"rts":1,"favs":0}
2017/09/21 22:00:42 Received event: {"messageid":33,"rts":-1,"favs":0}
2017/09/21 22:00:42 Received event: {"messageid":33,"rts":0,"favs":1}
2017/09/21 22:00:42 Received event: {"messageid":33,"rts":-1,"favs":0}
2017/09/21 22:00:47 Materializing 6 events from memory
```

As it is written in Go, the client code is quite verbose and at 212 lines won't fit into these pages. We might have a look at the *materialize* function though, because it's an interesting implementation of pushing the in-memory cache data structure down to our PostgreSQL table *twcache.counters*.

The in-memory cache structure looks like the following:

```
type Counter struct {
    MessageId int `json:"messageid"`
    Rts       int `json:"rts"`
    Favs      int `json:"favs"`
}

type Cache map[int]*Counter
```

And given such a data structure, we use the efficient Go default JSON marshaling facility to transform the cache elements and pass them all down to PostgreSQL as a single JSON object.

```
func materialize(db *sql.DB, cache Cache) error {
    ...

    js, err := json.Marshal(cache)

    if err != nil {
        log.Printf("Error while materializing cache: %s", err)
        return err
    }

    _, err = db.Query(q, js)

    ...
    return nil
}
```

The JSON object is then processed in a SQL query, that we find embedded in the Go code — it's the *q* string variable that is used in the snippet above in the expression *db.Query(q, js)*, where *js* is the JSON representation of the entirety of the cache data.

Here's the SQL query we use:

```
with rec as
 (
   select rec.*
     from json_each($1) as t,
          json_populate_record(null::twcache.counters, value) as rec
 )
 insert into twcache.counters(messageid, rts, favs)
      select messageid, rts, favs
        from rec
 on conflict (messageid)
   do update
         set rts  = counters.rts + excluded.rts,
             favs = counters.favs + excluded.favs
       where counters.messageid = excluded.messageid
```

In this query, we use the PostgreSQL json_populate_record function. This function is almost magical and it is described as such in the documentation:

> Expands the object in from_json to a row whose columns match the record type defined by base (see note below).
>
> Note: In *json_populate_record*, *json_populate_recordset*, *json_to_record* and *json_to_recordset*, type coercion from the JSON is "best effort" and may not result in desired values for some types. JSON keys are matched to identical column names in the target row type. JSON fields that do not appear in the target row type will be omitted from the output, and target columns that do not match any JSON field will simply be NULL.

The function allows transforming a JSON document into a full-blown relational tuple to process as usual in PostgreSQL. Here we use an implicit *lateral* construct that feeds the *json_populate_record()* function from the output of the *json_each()* function. We could have used the *recordset* variant, but we're discarding the Go cache key that repeats the *MessageId* here.

Then our SQL query uses the *insert into ... select ... on conflict do update* variant that we're used to by now.

Baring JSON tricks, the classic way to serialize a complex data structure targetting multiple rows is shown in the batch update example that follows this section.

It's important to note that coded as such, we can use the function to both materialize a full cache as fetched at startup, and to materialize the cache we build in-memory while receiving notifications.

The query used to fetch the initial value of the cache and set it again at startup is the following:

```
1 select messageid, rts, favs
2   from tweet.message_with_counters;
```

We use the view definition that we saw earlier to do the computations for us, and fill in our in-memory cache data structure from the result of the query.

The trigger processing has a cost of course, as we can see in the following

test:

```
CL-USER> (concurrency::concurrency-test 100 100 35)
Starting benchmark for updates
Updating took 8.428939 seconds, did 10000 rts

Starting benchmark for inserts
Inserting took 10.351908 seconds, did 10000 rts
```

Remember when reading those numbers that we can't compare them meaningfully anymore. We installed our trigger after insert on *tweet.activity*, which means that the update benchmark isn't calling any trigger whereas the insert benchmark is calling our trigger function 10,000 times in this test.

About the concurrency, notifications are serialized at commit time in the same way that the PostgreSQL commit log is serialized, so there's no extra work for PostgreSQL here.

Our cache maintenance server received 10,000 notifications with a JSON payload and then reported the cumulated figures to our cache table only once, as we can see from the logs:

```
2017/09/21 22:24:06 Received event: {"messageid":35,"rts":1,"favs":0}
2017/09/21 22:24:06 Received event: {"messageid":35,"rts":1,"favs":0}
2017/09/21 22:24:06 Received event: {"messageid":35,"rts":1,"favs":0}
2017/09/21 22:24:06 Received event: {"messageid":35,"rts":1,"favs":0}
2017/09/21 22:24:09 Materializing 1 events from memory
```

Having a look at the cache, here's what we have:

```
table twcache.counters;
```

```
 messageid |  rts   | favs
═══════════╪════════╪══════
         1 |  41688 |    0
         2 | 222690 |    0
         3 |  22000 |    0
        33 |     -4 |    8
         5 |   7000 |    0
         6 |  30000 |    0
        35 |  10000 |    0
(7 rows)
```

We can see the results of our tests, and in particular, the message with *ids* from 1 to 6 are in the cache as expected. Remember the rules we introduced earlier where the first thing we do when starting our cache maintenance service is to *reset* the cache from the real values in the database. That's

how we got those values in the cache; alter all, the cache service wasn't written when we ran our previous series of tests.

Limitations of Listen and Notify

It is crucial that an application using the PostgreSQL notification capabilities are capable of missing events. Notifications are only sent to connected client connections.

Any queueing mechanism requires that event accumulated when there's no worker connected are kept available until next connection, and replication is a special case of event queueing. It is not possible to implement queueing correctly with PostgreSQL *listen/notify* feature.

A cache maintenance service really is the perfect use case for this functionality, because it's easy to reset the cache at service start or restart.

Listen and Notify Support in Drivers

Support for listen and notify PostgreSQL functionality depends on the driver you're using. For instance, the Java JDBC driver documents the support at PostgreSQL Extensions to the JDBC API, and quoting their page:

> A key limitation of the JDBC driver is that it cannot receive asynchronous notifications and must poll the backend to check if any notifications were issued. A timeout can be given to the poll function, but then the execution of statements from other threads will block.

There's still a full-length class implementation sample, so if you're using Java check it out.

For Python, the Psycopg driver is the most popular, and Python asynchronous notifications supports advanced techniques for avoiding *busy looping*:

> A simple application could poll the connection from time to time to check if something new has arrived. A better strategy is to use some I/O completion function such as select() to sleep

> until awakened by the kernel when there is some data to read on the connection, thereby using no CPU unless there is something to read.

The Golang driver pq also supports notifications and doesn't require polling. That's the one we've been using this driver in our example here.

For other languages, please check the documentation of your driver of choice.

Batch Update, MoMA Collection

The Museum of Modern Art (MoMA) Collection hosts a database of the museum's collection, with monthly updates. The project is best described in their own words:

> MoMA is committed to helping everyone understand, enjoy, and use our collection. The Museum's website features 75,112 artworks from 21,218 artists. This research dataset contains 131,585 records, representing all of the works that have been accessioned into MoMA's collection and cataloged in our database. It includes basic metadata for each work, including title, artist, date made, medium, dimensions, and date acquired by the Museum. Some of these records have incomplete information and are noted as "not Curator Approved."

Using *git* and *git lfs* commands, it's possible to retrieve versions of the artist collection for the last months. From one month to the next, lots of the data remains unchanged, and some is updated.

```
1  begin;
2
3  create schema if not exists moma;
4
5  create table moma.artist
6   (
7     constituentid   integer not null primary key,
8     name            text not null,
9     bio             text,
10    nationality     text,
11    gender          text,
12    begin           integer,
```

```
   "end"           integer,
   wiki_qid        text,
   ulan            text
 );

\copy moma.artist from 'artists/artists.2017-05-01.csv' with csv header delimiter ','

commit;
```

Now that we have loaded some data, let's have a look at what we have:

```
select name, bio, nationality, gender
  from moma.artist
 limit 6;
```

Here are some of the artists being presented at the MoMA:

```
      name        │          bio          │ nationality │ gender
══════════════════╪═══════════════════════╪═════════════╪════════
 Robert Arneson   │ American, 1930 – 1992 │ American    │ Male
 Doroteo Arnaiz   │ Spanish, born 1936    │ Spanish     │ Male
 Bill Arnold      │ American, born 1941   │ American    │ Male
 Charles Arnoldi  │ American, born 1946   │ American    │ Male
 Per Arnoldi      │ Danish, born 1941     │ Danish      │ Male
 Danilo Aroldi    │ Italian, born 1925    │ Italian     │ Male
(6 rows)
```

Updating the Data

After having successfully loaded the data from May, let's say that we have received an update for June. As usual with updates of this kind, we don't have a *diff*, rather we have a whole new file with a new content.

A *batch update* operation is typically implemented that way:

- Load the new version of the data from file to a PostgreSQL table or a *temporary* table.
- Use the *update* command ability to use *join* operations to update existing data with the new values.
- Use the *insert* command ability to use *join* operations to insert new data from the *batch* into our target table.

Here's how to write that in SQL in our case:

```
begin;

```

```
create temp table batch
 (
        like moma.artist
   including all
 )
 on commit drop;

\copy batch from 'artists/artists.2017-06-01.csv' with csv header delimiter ','

with upd as
(
     update moma.artist
        set (name, bio, nationality, gender, begin, "end", wiki_qid, ulan)

          = (batch.name, batch.bio, batch.nationality,
             batch.gender, batch.begin, batch."end",
             batch.wiki_qid, batch.ulan)

       from batch

      where batch.constituentid = artist.constituentid

        and (artist.name, artist.bio, artist.nationality,
             artist.gender, artist.begin, artist."end",
             artist.wiki_qid, artist.ulan)
         <> (batch.name, batch.bio, batch.nationality,
             batch.gender, batch.begin, batch."end",
             batch.wiki_qid, batch.ulan)

  returning artist.constituentid
),
    ins as
(
    insert into moma.artist
         select constituentid, name, bio, nationality,
                gender, begin, "end", wiki_qid, ulan
           from batch
          where not exists
                (
                     select 1
                       from moma.artist
                      where artist.constituentid = batch.constituentid
                )
      returning artist.constituentid
)
select (select count(*) from upd) as updates,
       (select count(*) from ins) as inserts;

commit;
```

Our *batch update* implementation follows the specifications very closely.

The ability for the *update* and *insert* SQL commands to use *join* operations are put to good use, and the *returning* clause allows to display some statistics about what's been done.

Also, the script is careful enough to only update those rows that actually have changed thanks to using a *row comparator* in the update part of the *CTE*.

Finally, note the usage of an *anti-join* in the insert part of the *CTE* in order to only insert data we didn't have already.

Here's the result of running this *batch update* script:

```
BEGIN
CREATE TABLE
COPY 15186
 updates | inserts
═════════╪═════════
      35 |      21
(1 row)

COMMIT
```

An implicit assumption has been made in the creation of this script. Indeed, it considers the *constituentid* from MoMA to be a reliable primary key for our data set. This assumption should, of course, be checked before deploying such an update script to production.

Concurrency Patterns

While in this solution the update or insert happens in a single query, which means using a single *snapshot* of the database and a within a transaction, it is still not prevented from being used concurrently. The tricky case happens if your application were to run the query above twice at the same time.

What happens is that as soon as the concurrent sources contain some data for the same *primary key*, you get a *duplicate key* error on the insert. As both the transactions are concurrent, they are seeing the same *target* table where the new data does not exists, and both will conclude that they need to *insert* the new data into the *target* table.

There are two things that you can do to avoid the problem. The first thing is to make it so that you're doing only one *batch update* at any time, by

building your application around that constraint.

A good way to implement that idea is with a manual *lock* command as explain in the explicit locking documentation part of PostgreSQL:

```
LOCK TABLE target IN SHARE ROW EXCLUSIVE MODE;
```

That *lock level* is not automatically acquired by any PostgreSQL command, so the only way it helps you is when you're doing that for every transaction you want to serialize. When you know you're not at risk (that is, when not playing the *insert or update* dance), you can omit that *lock*.

Another solution is using the new in PostgreSQL 9.5 *on conflict* clause for the *insert* statement.

On Conflict Do Nothing

When using PostgreSQL version 9.5 and later, it is possible to use the *on conflict* clause of the *insert* statement to handle concurrency issues, as in the following variant of the script we already saw. Here's a *diff* of the first update script and the second one, that handles concurrency conflicts:

```
--- artists.update.sql  2017-09-07 23:54:07.000000000 +0200
+++ artists.update.conflict.sql 2017-09-08 12:49:44.000000000 +0200
@@ -5,11 +5,11 @@
         like moma.artist
    including all
  )
  on commit drop;

-\copy batch from 'artists/artists.2017-06-01.csv' with csv header delimiter ','
+\copy batch from 'artists/artists.2017-07-01.csv' with csv header delimiter ','

 with upd as
 (
      update moma.artist
         set (name, bio, nationality, gender, begin, "end", wiki_qid, ulan)
@@ -41,10 +41,11 @@
                  (
                       select 1
                         from moma.artist
                        where artist.constituentid = batch.constituentid
                  )
+    on conflict (constituentid) do nothing
       returning artist.constituentid
 )
 select (select count(*) from upd) as updates,
```

```
26          (select count(*) from ins) as inserts;
27
```

Notice the new line *on conflict (constituentid) do nothing*. It basically implements what it says: if inserting a new row causes a conflict, then the operation for this row is skipped.

The conflict here is a *primary key* or a *unique* violation, which means that the row already exists in the target table. In our case, this may only happen because a concurrent query just inserted that row while our query is in flight, in between its lookup done in the *update* part of the query and the *insert* part of the query.

An Interview with Kris Jenkins

Kris Jenkins is a successful startup cofounder turned freelance functional programmer, and open-source enthusiast. He mostly works on building systems in Elm, Haskell & Clojure, improving the world one project at a time.

Kris Jenkins is the author of the YeSQL library, and approach that we've seen in this book in the chapter Writing SQL queries.

As a full stack developer, how do you typically approach concurrency behavior in your code? Is it a design-time task or more a scaling and optimizing aspect of your delivery?

> *I try to design for correctness & clarity, rather than performance. You'll never really know what your performance and scaling hotspots will be until you've got some real load on the system, but you'll always want correctness and clarity. That mindset dictates how I approach concurrency problems. I worry about things like transaction boundaries up-front, before I write a line of code because I know that if I get that wrong, it's going to bite me at some point down the line. But for performance, I'll tend to wait and see. Real world performance issues rarely crop up where you predict — it's better to observe what's really happening. Same with scaling issues — you might think you know which parts of the system will be in high demand,*

but reality will often surprise you. But if you focus on getting the system clear — readable and maintainable — it's easier to adapt the design for version two.

Of course, there are exceptions to that. If I knew a certain system would have a million users on day one, obviously that would change things. But even then, I'd code up a naive-but-correct prototype.

The point being, concurrency has two sides — "is it right?" and "is it fast?" — and I worry about the first one first. As Paul Phillips rightly said, performance is tail, correctness is dog. You don't let the tail wag the dog.

How much impact would the choice of a stack would have on your approach to concurrency behavior? You've been doing lots of Clojure and Haskell, and those are pretty different from the more classic PHP or Python. Do they help to implement concurrency correct code?

Definitely. They're a huge help. The reason Clojure exists is to bring some clarity to how we deal with the effect of time in programming. Clojure's key insight is that time doesn't just complicate concurrent database transactions, but nearly every aspect of programming. Concurrency is, "what happens if someone else stomps over my data at the same time?" Mutability is, "what happens if someone else stomps over my data, or if I stomp on it myself?" Languages with immutable data structures, like Clojure, ask, "Why don't we just eliminate that whole problem?"

So Clojure is designed from the ground up to eliminate the effect of time from programming completely, and then only bring it back in when you really need it. By default there is no concurrency, there are no competing timelines, and then if you really need to bring it back you get great support for doing so. You opt-in to concurrency problems, carefully and with great support. That both frees you up from worrying about concurrency and makes you very mindful of it.

Haskell I'd say takes that even further. It doesn't just make you suspicious of the side-effects time is having on your code,

but just about all side effects. Haskell's been ferocious about side effects for... I guess twenty years now... and it's still an active area of research to beat them down even harder.

So both languages beat out side effects and then gradually bring them back in, with controls. And what controls do we see for concurrency? For the most part, it's not the low-level locks and semaphores of C and Java, but the higher-level ideas we love from databases, like repeatable reads (immutability) and ACID transactions (software transactional memory).

When using PostgreSQL in your application stacks, which role do you assign to it? Is it more of a library, framework, storage engine, processing workhouse, or something else entirely?

It has two key roles for me. First, as the storage engine. It's the golden record of what our system knows. Every important fact should be there. That probably makes the database the most precious part of the system, but that's okay. Data is precious.

The other role is much more abstract. I use the database a design tool. There's something great about the relational mindset that encourages you to think about data in itself, separate from how it's used. You model the data so it takes its own real shape, rather than the shape today's task wants it to have.

By way of contrast, I think one of the downsides of test-driven-development is in some corners it's encouraged people to think of their data as a kind of black box, where only the way it's used today gets to drive the data implementation. Too often I've seen that lead to big painful rewrites when the data outgrows the features.

The mindset of making data primary, and today's use-case secondary, is invaluable if you want a system to grow well. And that's something Codd figured out decades ago.

I was lucky enough early in my career to get a job with a financial database company that really only existed because they had a better data model than all their competitors. The whole product fell out of the fact that they figured out a better schema than everyone else, so they could do many things their competitors

struggled with, without breaking a sweat. They taught me early on that if you get your data model right, every feature is easier. You can get things right without trying while you competitors firefight their mistakes.

When using PostgreSQL, do you pick the default isolation level or do you have a specific approach to picking the right isolation level depending on the task you're implementing?

Ha, boring answer here — I stick to the default! I don't think I've changed it more than a couple of times, for a couple of extremely specific cases.

8

Closing Thoughts

I have written *Mastering PostgreSQL in Application Development* so that as a developer, you may think of SQL as a full-blown programming language. Some of the problems that we have to solve as developers are best addressed using SQL.

Not just any SQL will do: ***PostgreSQL is the world's most advanced open source database***. I like to say that *PostgreSQL is YeSQL* as a pun, which compares it favorably to many NoSQL solutions out there. PostgreSQL delivers the whole SQL experience with advanced data processing functionality and document-based approaches.

We have seen many SQL features — I hope many you didn't know before. Now you can follow the *one resultset, one query* mantra, and maintain your queries over the entirety of their life cycles: from specification to testing, including code review and rewrite.

Of course your journey into *Mastering PostgreSQL in Application Development* is only starting. Writing code is fun. Have fun writing SQL!

> *Knowledge is of no value unless you put it into practice.*
>
> — ***Anton Chekhov***