SECOND EDITION

The Art of PostgreSQL

Turn Thousands of Lines of Code into Simple Queries

by Dimitri Fontaine

Contents

Part I

Preface

As a developer, *The Art of PostgreSQL* is the book you need to read in order to get to the next level of pro ciency.

Af er 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 ll 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 dominat . If you've chosen the right data structur and orga-

nized things well, the algorithms will almost always be self-evident.

Data structur , not algorithms, are central to programming.

— Rob Pike

About…

About the Book

This book is intended for developers working on applications that use a database

server. The book speci cally addresses the PostgreSQL RDBMS: it actually is the world's most advanced Open Source database, just like it says in the tagline on the

o cial website. By the end of this book you'll know why, and you'll agree!

I wanted to write this book af er having worked with many customers who were

making use of only a fraction of what SQL and PostgreSQL are capable of deliv-

ering. 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 e cient SQL query, then in some cases they would nonetheless not know how to integrate a raw SQL query in their code base.

Integrating a SQL query and thinking about SQL as code means using the same

advanced tooling that we use when using other programming languages: version-

ing, automated testing, code reviewing, and deployment. Really, this is more

about the developer’s work ow than the SQL code itself…

In this book, you will learn best practices that help with integrating SQL into

your own work ow, 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

pervaded by the reassuring illusion that programs

are just devic like any others, the only difference admitted being

that their manufacture might require a new type of craftsmen, viz.

programmers. From there it

only a small step to measuring “pro-

grammer productivity” in terms of “number of lin of code pro-

duced per month”. Th

a very costly measuring unit because it

encourag the writing of insipid code, but today I am less interested

in how foolish a unit it

from even a pure business point of view.

My point today

that, if we wish to count lin of code, we should

not regard them

“lin produced” but

“lin spent”: the current

conventional wisdom so foolish

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 Sof ware 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 man-

ager, sof ware architect, database architect, and database administrator at di fer-

ent 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 execu-

tive 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, Kris Jenkins, functional programmer and author of the YeSQL library for Clojure, and Craig Kerstiens, head of Could at Citus Data.

Having insights from SQL users from many di ferent backgrounds has been valu-

able 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!

About the organisation of the books

Each part of “The Art of PostgreSQL” can be read on its own, or you can read

this book from the rst to the last page in the order of the parts and chapters

therein. A great deal of thinking have been put in the ordering of the parts, so

that reading “The Art of PostgreSQL” in a linear fashion should provide the best

experience.

The skill progression throughout the book is not linear. Each time a new SQL

concept is introduced, it is presented with simple enough queries, in order to

make it possible to focus on the new notion. Then, more queries are introduced

to answer more interesting business questions.

Complexity of the queries usually advances over the course of a given part, chap-

ter af er chapter. Sometimes, when a new chapter introduces a new SQL concept, complexity is reset to very simple queries again. That’s because for most

people, learning a new skill set does not happen in a linear way. Having this kind

of di culty organisation also makes it easier to dive into a given chapter out-of-

order.

Here’s a quick breakdown of what each chapter contains:

Part 1, Preface

You’re reading it now, the preface is a presentation of the book and what to expect

from it.

Part 2, Introduction

The introduction of this book intends to convince application developers such

as you, dear reader, that there’s more to SQL than you might think. It begins

with a very simple data set and simple enough queries, that we compare to their

equivalent Python code. Then we expand from there with a very important trick

that’s not well known, and a pretty advanced variation of it.

Part 3, Writting SQL Queries

The third part of the book covers how to write a SQL query as an application

developer. We answer several important questions here:

• Why using SQL rather than your usual programming language?

• How to integrate SQL in your application source code?

• How to work at the SQL prompt, the psql REPL?

• What’s an indexing strategy and how to approach indexing?

A simple Python application is introduced as a practical example illustrating the

di ferent answers provided. In particular, this part insists on when to use SQL

to implement business logic.

Part 3 concludes with an interview with Yohan Gabory, author of a French book

that teaches how to write advanced web application with Python and Django.

Part 4, SQL Toolbox

The fourth part of “The Art of PostgreSQL” introduces most of the SQL con-

cepts that you need to master as an application developer. It begins with the

basics, because you need to build your knowledge and skill set on-top of those

foundations.

Advanced SQL concepts are introduced with practical examples: every query

refers to a data model that’s easy to understand, and is given in the context of a

“business case”, or “user story”.

This part covers SQL clauses and features such as ORDER BY and k-NN sorts,

the GROUP BY and HAVING clause and GROUPING SETS, along with clas-

sic and advanced aggregates, and then window functions. This part also covers

the infamous NULL, and what's a relation and a join.

Part 5 concludes with an interview with Markus Winand, author of "SQL Perfor-

mance explained" and http://use-the-index-luke.com. Markus is a master of the SQL standard and he is a wizard on using SQL to enable fast application delivery

and solid run-time performances!

Part 5, Data Types

The f h part of this book covers the main PostgreSQL data types you can use

and bene t from as an application developer. PostgreSQL is an ORDBMS:

Object-Oriented Relation Database Manager. As a result, data types in Post-

greSQL are not just the classics numbers, dates, and text. There's more to it,

and this part covers a lot of ground.

Part 5 concludes with an interview with Grégoire Hubert, author of the POMM

project, which provides developers with unlimited access to SQL and database

features while proposing a high-level API over low-level drivers.

Part 6, Data Modeling

The sixth part of “The Art of PostgreSQL” covers the basics of relational data

modeling, which is the most important skill you need to master as an application

developer. Given a good database model, every single SQL query is easy to write,

things are kep logical, and data is kept clean. With a bad design… well my guess is

that you’ve seen what happens with a not-great data model already, and in many

cases that’s the root of developers’ disklike for the SQL language.

This part comes late in the book for a reason: without knowledge of some of the

advanced SQL facilities, it’s hard to anticipate that a data model is going to be easy enough to work with, and developers then tend to apply early optimizations to

the model to try to simplify writing the code. Well, most of those *optimizations*

are detrimental to our ability to bene t from SQL.

Part 6 concludes with an interview with Álvaro Hernández Tortosa, who built

the ToroDB project, a MongoDB replica solution based on PostgreSQL! His take on relation database modeling when compared to NoSQL and document

based technologies and APIs is the perfect conclusion of the database modeling part.

Part 7, Data Manipulation and Concurrency Control

The seventh part of this book covers DML and concurrency, the heart of any live

database. DML stands for “Data Modi cation Language”: it’s the part of SQL

that includes INSERT, UPDATE, and DELETE statements.

The main feature of any RDBMS is how it deals with concurrent access to a single

data set, in both reading and writing. This part covers isolation and locking,

computing and caching in SQL complete with cache invalidation techniques,

and more.

Part 7 concludes with an interview with Kris Jenkins, a functional programmer

and open-source enthusiast. He mostly works on building systems in Elm,

Haskell & Clojure, improving the world one project at a time, and he’s is the

author of the YeSQL library.

Part 8, PostgreSQL Extensions

The eighth part of “The Art of PostgreSQL” covers a selection of very useful Post-

greSQL Extensions and their impact on simplifying application development

when using PostgreSQL.

We cover auditing changes with hstore, the pg_trgm extension to implement

auto-suggestions and auto-correct in your application search forms, user-de
ned

tags and how to e ciently use them in search queries, and then we use ip4r for

implementing geolocation oriented features. Finally, hyperlolog is introduced

to solve a classic problem with high cardinality estimates and how to
combine

them.

Part 8 concludes with an interview with Craig Kerstiens who heads the Cloud

team at Citus Data, af er having been involved in PostgreSQL support at

Heroku. Craig shares his opinion about using PostgreSQL extensions when

deploying your application using a cloud-based PostgreSQL solution.

Part II

Introduction

1

Structured Query Language

SQL stands for *Structured Query Language*; the term de nes a declarative
pro-

gramming 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 de nes

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 speci c 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 whether he considers h code or h data structur

more important. Bad programmers worry about the code. Good

programmers worry about data structur and their relationships.

— *Lin*

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 Sof ware Module of Any Type?

SQLite

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 iTun , every Dropbox client, every TurboTax and Quick-

Books, PHP and Python, most television sets and set-top cable box ,

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 pro-

cessing 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 sof ware 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 Sys-

tems 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* le which is actually a *CSV* le 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 gures 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 de nition, and once the data is loaded we then trans-

form it into a proper SQL data type, thanks to *alter table* commands.

1

begin;

2

3

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

```
\copy factbook from 'factbook.csv' with delimiter E'\t' null ''

alter table factbook
alter shares
type bigint
```

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

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

commit;
```

We use the PostgreSQL copy functionality to stream the data from the CSV

le into our table. The \copy variant is a *psql* speci c command and initiates *client/server* streaming of the data, reading a local le 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 rst 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 rst day of the month gives us the rst 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 speci ed number of digits

• *L*, currency symbol (uses locale)

• *G*, group separator (uses locale)

Other template patterns for numeric formatting are available — see the Post-

greSQL 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 ll it in with either matching data

or a zero gure for days without data in our *factbook*.

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

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

import sys
import psycopg2
import psycopg2.extras
```

```
from calendar import Calendar


CONNSTRING = "dbname=yesql application_name=factbook"



def fetch_month_data(year, month):
"Fetch a month of data from the database"
date = "%d-%02d-01" % (year, month)
sql = """
select date, shares, trades, dollars
from factbook

```

```
where date >= date %s
and date < date %s + interval '1 month'
order by date;
"""
pgconn = psycopg2.connect(CONNSTRING)
curs = pgconn.cursor()
curs.execute(sql, (date, date))

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

```
res[date] = (shares, trades, dollars)

return res

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

cal = Calendar()
```

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

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

```
shares, trades, dollars = 0, 0, 0
```

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

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

list_book_for_month(year, month)
```

In this implementation, we use the above SQL query to fetch our result set, and moreover to store it in a dictionary. The dict’s key is the day of the month, so we

can then loop over a calendar’s list of days and retrieve matching data when we

have it and install a default result set (here, zeroes) when we don’t have anything.

Below is the output when running the program. As you can see, we opted for an

output similar to the *psql* output, making it easier to compare the e fort 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 |

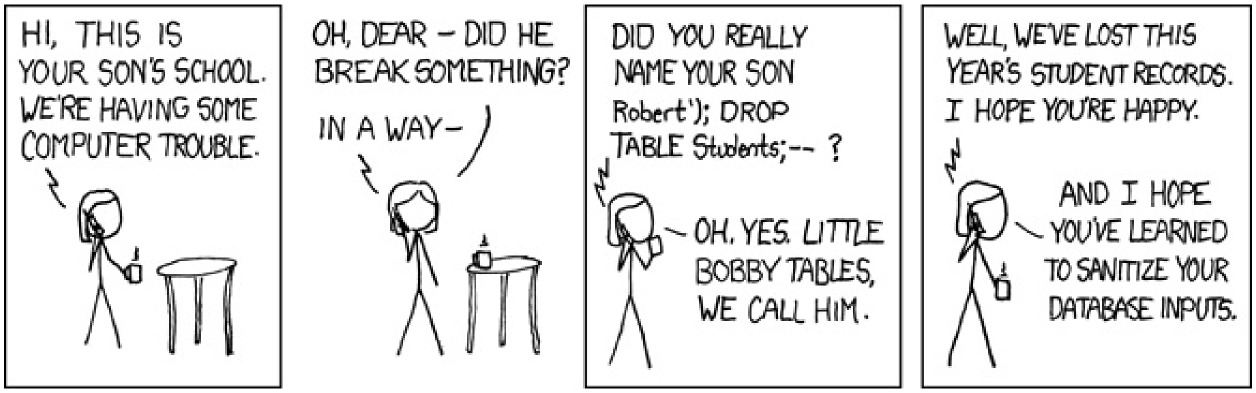
43613734358

2017-02-28 |

1455597403 |

4789769 |

57874495227


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

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 Tabl* .

Figure 1.1: Exploits of a Mom

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

The PostgreSQL protocol is fully documented and you can read more about *ex-*

tended 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 under-

stand how to send SQL query parameters separately from the main SQL query

text; this is a reliable way to never have to worry about *SQL injection* problems ever again.

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

library, ORM or another tooling that would do that. When building SQL query

strings that way, you open your application code to serious security risk for no

reason.

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

SQL queries right from the beginning.

When using Psycopg the SQL query parameters are interpolated in the SQL

query string at the client level. It means you need to trust Psycopg to protect

you from any attempt at SQL injection, and we could be more secure than that.

PostgreSQL protocol: server-side prepared state-

ments

It is possible to send the query string and its arguments separately on the wire

by using server-side prepared statements. This is a pretty common way to do

it, mostly because PQexecParams isn’t well known, though it made its debut in

PostgreSQL 7.4, released November 17, 2003. To this day, a lot of PostgreSQL

drivers still don’t expose the PQexecParams facility, which is unfortunate.

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

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

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

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

We then get the same result as before, when using our rst version of the

Python

program.

Now, while it’s possible to use the prepare and execute SQL commands directly in your application code, it is also possible to use it directly at the PostgreSQL

protocol level. This facility is named Extended Query and is well documented.

Reading the documentation about the protocol implementation, we see the fol-

lowing bits. First the PARSE message:

In the extended protocol, the frontend rst sends a Parse message,

which contains a textual query string, optionally some information

about data types of parameter placeholders, and the name of a des-

tination prepared-statement object […]

Then, the BIND message:

Once a prepared statement exists, it can be readied for execution us-

ing a Bind message. […] The supplied parameter set must match

those needed by the prepared statement.

Finally, to receive the result set the client needs to send a third message, the EXE-

CUTE message. The details of this part aren’t relevant now though.

It is very clear from the documentation excerpts above that the query string

parsed by PostgreSQL doesn’t contain the parameters. The query string is

sent in the BIND message. The query parameters are sent in the EXECUTE

message. When doing things that way, it is impossible to have SQL
injections.

Remember: SQL injection happens when the SQL parser is fooled into
believ-

ing that a parameter string is in fact a SQL query, and then the SQL engine
goes

on and executes that SQL statement. When the SQL query string lives in your

application code, and the user-supplied parameters are sent separately on the
net-

work, there’s no way that the SQL parsing engine might get confused.

The following example uses the asyncpg PostgreSQL driver. It’s open source
and the sources are available at the MagicStack/asyncpg repository, where
you can browse the code and see that the driver implements the PostgreSQL
protocol

itself, and uses server-side prepared statements.

This example is now safe from SQL injection by design, because the server-
side

prepared statement protocol sends the query string and its arguments in
separate

protocol messages:

1

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

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


async def fetch_month_data(year, month):
    "Fetch a month of data from the database"
```

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

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

await pgconn.close()

return res
```

Then, the Python function call needs to be adjusted to take into account the coroutine usage we’re now making via asyncio.

The function list_book_for_month now begins with the following lines:

```
def list_book_for_month(year, month):
```

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

The rest of it is as before.

Back to Discovering SQL

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

```
select cast(calendar.entry as date) as date,
coalesce(shares, 0) as shares,
coalesce(trades, 0) as trades,
to_char(
coalesce(dollars, 0),
'L99G999G999G999'
```

```
) as dollars
from /*
* Generate the target month's calendar then LEFT JOIN
* each day against the factbook dataset, so as to have
* every day in the result set, whether or not we have a
* book entry for the day.
*/
generate_series(date :'start',
date :'start' + interval '1 month'
- interval '1 day',
interval '1 day'
)
as calendar(entry)
left join factbook
on factbook.date = calendar.entry
order by date;
```

In this query, we use several basic SQL and PostgreSQL techniques that you might be discovering for the rst 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_seri ()* 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 rst day of the month as a single parameter in the

query.

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

day'.

• The *cast(calendar.entry*

date) expression transforms the generated *cal-*

endar.entry, which is the result of the *generate_seri ()* function call into the *date* data type.

We need to *cast* here because the *generate_seri ()_ 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*, *shar* , *trad* , and *dollars*) are lled in with *NULL* values instead.

• COALESCE returns the rst of its arguments that is not null.

So the expression *coalesce(shar , 0)*

shar 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 lled 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

nd 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 e cient 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 di ference for each day

of the result. More speci cally, 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 di ference” example because it’s both

a classic analytics need, though mostly in marketing circles maybe, and because

in my experience the rst reaction of a developer will rarely be to write a SQL

query doing all the math.

Also, computing weeks is another area in which the calendar we have isn’t very

helpful, but for PostgreSQL taking care of the task is as easy as spelling the word

week:

```
with computed_data as
(
select cast(date as date) as date,
to_char(date, 'Dy') as day,
coalesce(dollars, 0) as dollars,
lag(dollars, 1)
over(
partition by extract('isodow' from date)
order by date
)
as last_week_dollars
from /*
* Generate the month calendar, plus a week before
```

```
* so that we have values to compare dollars against
* even for the first week of the month.
*/
generate_series(date :'start' - interval '1 week',
date :'start' + interval '1 month'
- interval '1 day',
interval '1 day'
)
as calendar(date)
left join factbook using(date)
)
select date, day,
to_char(
coalesce(dollars, 0),
'L99G999G999G999'
```

```sql
) as dollars,
case when dollars is not null
and dollars <> 0
then round(
100.0
* (dollars - last_week_dollars)
/ dollars
, 2)
end
as "WoW %"
from computed_data
```

```
where date >= date :'start'
order by date;
```

To implement this case in SQL, we need *window functions* that appeared in the SQL standard in 1992 but are still of en skipped in SQL classes. The last thing

executed in a SQL statement are *windows functions*, well af er *join* operations and *where* clauses. So if we want to see a full week before the rst 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 di ference 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 ex-

ample.

2

Software Architecture

Our rst use case in this book allowed us to compare implementing a simple

feature in Python and in SQL. Af er 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 sof ware

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 lif ing yields a net gain in performance

characteristics too, mostly because round-trip times and latency along with mem-

ory and bandwidth resources usage depend directly on the size of the result sets.

The Art Of PostgreSQL, Volume 1 focuses on teaching SQL idioms, both the ba-

sics and some advanced techniques too. It also contains an approach to database modeling, normalization, and denormalization. That said, it does not address

sof ware 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 ap-

plication 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 sof ware architec-

ture is to handle concurrent access to live data that is manipulated by sev-

eral applications, or several parts of an application.

Typically we will nd the user-side parts of the application, a front-o ce

and a user back-o ce with a di ferent set of features depending on the user

role, including some kinds of reporting (accounting, nance, analytics),

and of en 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 transac-

tion. 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 lef 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 o fered 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 nd the most e cient 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 de nes 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 de ned 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 o fers a deep coverage for composite data types and documents processing right within the language, with clean semantics.

So when designing your sof ware 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 pro-

gramming language in modern application development, it only addresses the

PostgreSQL implementation of the SQL standard. That choice is down to sev-

eral factors, all consequences of PostgreSQL truly being *the world's most ad-*

vanced 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 mail-

ing lists for all discussions, contributions, and decisions, and the project

goes as far as self-hosting all requirements in order to avoid being in u-

enced 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 nd, 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 o fering di ferent trade-o fs in terms of op-

erations, guarantees, query languages and APIs, I would argue that Post-

greSQL 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 nd.

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 func-

tions 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 o fers a very di ferent 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.

Af er having spent some time as a developer using PostgreSQL, then as a

Post-

greSQL contributor and consultant, nowadays I can very easily nd 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 o fer 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 ne manual. No really… this one is

ne.

3

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 al-

ready 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 con gure it in a

friendly way.

You might prefer a more visual tool such as pgAdmin or OmniDB; the key here is to be able to easily edit SQL queries, run them, edit them in order to x 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 les. Check out the toc.txt le at the top of the

les tree, it contains a detailed table of contents and the list of les found in each section, such as in the following example:

2 Introduction

2 Structured Query Language

2.1 Some of the Code is Written in SQL

2.2 A First Use Case

2.3 Loading the Data Set

02-intro/02-usecase/03_01_factbook.sql

2.4 Application Code and SQL

02-intro/02-usecase/04_01.sql

02-intro/02-usecase/04_02_factbook-month.py

2.5 A Word about SQL Injection

2.6 PostgreSQL protocol: server-side prepared statements

02-intro/02-usecase/06_01.sql

02-intro/02-usecase/06_02.sql

2.7 Back to Discovering SQL

02-intro/02-usecase/07_01.sql

2.8 Computing Weekly Changes

02-intro/02-usecase/08_01.sql

3 Software Architecture

3.1 Why PostgreSQL?

02-intro/03-postgresql/01_01.sql

3.2 The PostgreSQL Documentation

4 Getting Ready to read this Book

02-intro/04-postgresql/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.

Part III

Writing Sql Queries

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 xed version of the query.

Before we go into details about the speci cs of those concerns, it might be a good

idea to review how SQL actually helps you write sof ware, 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 speci c code base.

Af er 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.

4

Business Logic

Where to maintain the *business logic* can be a hard question to answer. Each

application may be di ferent, and every development team might have a di ferent

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 speci cs, 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:

```sql
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 lef 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 speci c 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
join genre using(genreid)
```

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 im-

plementing 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:

```
select name,
milliseconds * interval '1 ms' as duration,
pg_size_pretty(bytes) as bytes
from track
where albumid = 193
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 rst de ne 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'
```

```
7
group by album
8
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 speci c 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
from datetime import timedelta

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


class Model(object):
tablename = None
columns = None

@classmethod
```

```
def buildsql(cls, pgconn, **kwargs):
if cls.tablename and kwargs:
cols = ", ".join(['"%s"' % c for c in cls.columns])
qtab = '"%s"' % cls.tablename
```

```
sql = "select %s from %s where " % (cols, qtab)
for key in kwargs.keys():
sql += "\"%s\" = '%s'" % (key, kwargs[key])
if DEBUGSQL:
print(sql)
return sql
```

```
@classmethod
def fetchone(cls, pgconn, **kwargs):
if cls.tablename and kwargs:
sql = cls.buildsql(pgconn, **kwargs)
curs = pgconn.cursor(cursor_factory=psycopg2.extras.DictCursor)
curs.execute(sql)
result = curs.fetchone()
if result is not None:
return cls(*result)
```

```
@classmethod
def fetchall(cls, pgconn, **kwargs):
if cls.tablename and kwargs:
sql = cls.buildsql(pgconn, **kwargs)
curs = pgconn.cursor(cursor_factory=psycopg2.extras.DictCursor)
curs.execute(sql)
resultset = curs.fetchall()
if resultset:
return [cls(*result) for result in resultset]


```

```
class Artist(Model):
tablename = "artist"
columns = ["artistid", "name"]

def __init__(self, id, name):
self.id = id
self.name = name


class Album(Model):
tablename = "album"
```

```
columns = ["albumid", "title"]

def __init__(self, id, title):
self.id = id
self.title = title
self.duration = None


class Track(Model):
tablename = "track"
columns = ["trackid", "name", "milliseconds", "bytes", "unitprice"]

```

```
def __init__(self, id, name, milliseconds, bytes, unitprice):
self.id = id
self.name = name
self.duration = milliseconds
self.bytes = bytes
self.unitprice = unitprice



if __name__ == '__main__':
if len(sys.argv) > 1:
pgconn = psycopg2.connect(PGCONNSTRING)
```

```
artist = Artist.fetchone(pgconn, name=sys.argv[1])

for album in Album.fetchall(pgconn, artistid=artist.id):
ms = 0
for track in Track.fetchall(pgconn, albumid=album.id):
ms += track.duration

duration = timedelta(milliseconds=ms)
print("%25s: %s" % (album.title, duration))
else:
print('albums.py <artist name>')
```

Now the result of this code is as following:

```
$ ./albums.py "Red Hot Chili Peppers"
Blood Sugar Sex Magik: 1:13:57.073000
By The Way: 1:08:49.951000
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 ob-
ject model will insist on hydrating the intermediate objects with as much infor-
mation 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 de nes four isolation levels and PostgreSQL implements

three of them, leaving out *dirty reads*. The isolation level determines which side e fects 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 di ferent counts at di ferent points in your transaction.

```
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 di ferent 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 di ferent 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 di ferent artist to x 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

E ciency 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 solu-

tion, the maintenance burden it then represents (like the likelihood of bug xes, the complexity of xing 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, net-

work, 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 ve 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 e ciency.

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 e cient to use, and for no good reason. So, instead, we are

going to de ne 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 rst 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 e cient, 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 bene ts 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
(
in name text,
out album text,
out duration interval
)
returns setof record
language plpgsql
as $$
declare
rec record;
begin
```

```
for rec in select albumid
from album
join artist using(artistid)
where album.name = get_all_albums.name
loop
select title, sum(milliseconds) * interval '1ms'
into album, duration
from album
left join track using(albumid)
where albumid = record.albumid
```

```
group by title
order by title;
```

```
return next;
end loop;
end;
$$;
```

What we see here is basically a re-enactment of everything we said was wrong to

do in our application code example. The main di ference 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 e cient, the default

should be SQL.

Where to Implement Business Logic?

We saw di ferent 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 rst solution is both incorrect and ine cient, 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 ve 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 con-

current users browsing your album collection do you want to be able to serve?

When doing ve 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 in-

vest in an extra layer of caching architecture in front of your APIs, wouldn't it

be better to write smarter and more e cient SQL?

As for stored procedures, a lot has already been said. Using them allows the de-

velopers to build a data access API in the database server and to maintain it in a transactional way with the database schema: PostgreSQL implements transac-

tions 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.

5

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* rst.

That's this little le 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 rst, it didn’t support PostgreSQL, so I used

the SQLite data output, which nicely ts into a small enough data le. Nowadays

you will nd a PostgreSQL backup le 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 rst encounter with our dataset.

Here’s a truncated output from the pgloader run (edited so that it can t 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 x a badly de ned 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 x 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 nd 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:

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

5

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 les. We will look at the Python side of

things in a moment.

The artist.sql le 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 les in our source tree allows us to version control them with git,

write comments when necessary, and also copy and paste the les between your

application’s directory and the interactive psql shell.

In the case of our artist.sql le, we see the use of the *anosql* facility to name variables and we use limit :n. Here’s how to bene t 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 le 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 le. 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
/*
* the lateral left join implements a nested loop over
* the genres and allows to fetch our Top-N tracks per
```

```
* genre, applying the order by desc limit n clause.
*

```

```
* here we choose to weight the tracks by how many
* times they appear in a playlist, so we join against
* the playlisttrack table and count appearances.
*/
(
select track.name, track.albumid, count(playlistid)
from track
left join playlisttrack using (trackid)
where track.genreid = genre.genreid
group by track.trackid
order by count desc
limit :n
)
/*
* the join happens in the subquery's where clause, so
* we don't need to add another one at the outer join
* level, hence the "on true" spelling.
*/
ss(name, albumid, count) on true
join album using(albumid)
join artist using(artistid)
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 nd it easier to read and gure out the equivalent code we would have

had to write in Python. The main reason why writing moderately complex SQL

for this listing is e ciency.

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 e cient than the *lateral* method. We will cover how to read an *explain plan* in a later chapter, and that’s how to gure out the most e cient 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']:

queries = anosql.load_queries('postgres', sql)

if self.queries:
```

```
for qname in queries.available_queries:

self.queries.add_query(qname, getattr(queries, qname))

else:

self.queries = queries
```

```
def genre_list(self):

return self.queries.tracks_by_genre(self.pgconn)

def genre_top_n(self, n):

return self.queries.genre_top_n(self.pgconn, n=n)

def artist_by_albums(self, n):

return self.queries.top_artists_by_album(self.pgconn, n=n)

def album_details(self, albumid):

return self.queries.list_tracks_by_albumid(self.pgconn, id=albumid)

def album_by_artist(self, artist):

return self.queries.list_albums_by_artist(self.pgconn, name=artist)

class printer(object):

"print out query result data"

def __init__(self, columns, specs, prelude=True):

"""COLUMNS is a tuple of column titles,

Specs an tuple of python format strings
```

```
"""

self.columns = columns

self.specs = specs

self.fstr = " | ".join(str(i) for i in specs)

if prelude:

print(self.title())

print(self.sep())

def title(self):

return self.fstr % self.columns

def sep(self):

s = ""

for c in self.title():

s += "+" if c == "|" else "-"

return s

def fmt(self, data):

return self.fstr % data

class cdstore(object):

"""Our cdstore command line application. """

def __init__(self, argv):

self.db = chinook()
```

```
parser = argparse.ArgumentParser(
description='cdstore utility for a chinook database',
usage='cdstore <command> [<args>]')
subparsers = parser.add_subparsers(help='sub-command help')
genres = subparsers.add_parser('genres', help='list genres')
genres.add_argument('--topn', type=int)
```

```
genres.set_defaults(method=self.genres)
artists = subparsers.add_parser('artists', help='list artists')
artists.add_argument('--topn', type=int, default=5)
artists.set_defaults(method=self.artists)
albums = subparsers.add_parser('albums', help='list albums')
albums.add_argument('--id', type=int, default=None)
albums.add_argument('--artist', default=None)
albums.set_defaults(method=self.albums)
args = parser.parse_args(argv)
args.method(args)
def genres(self, args):
"List genres and number of tracks per genre"
if args.topn:
```

```
p = printer(("Genre", "Track", "Artist"),

("%20s", "%20s", "%20s"))

for (genre, track, artist) in self.db.genre_top_n(args.topn):

artist = artist if len(artist) < 20 else "%s…" % artist[0:18]

print(p.fmt((genre, track, artist)))

else:

p = printer(("Genre", "Count"), ("%20s", "%s")) for row in
self.db.genre_list():

print(p.fmt(row))

def artists(self, args):

"List genres and number of tracks per genre"

p = printer(("Artist", "Albums"), ("%20s", "%5s")) for row in
self.db.artist_by_albums(args.topn):

print(p.fmt(row))

def albums(self, args):

# we decide to skip parts of the information here

if args.id:

p = printer(("Title", "Duration", "Pct"),

("%25s", "%15s", "%6s"))

for (title, ms, s, e, pct) in self.db.album_details(args.id):

title = title if len(title) < 25 else "%s…" % title[0:23]
```

```
print(p.fmt((title, ms, pct)))

elif args.artist:

p = printer(("Album", "Duration"), ("%25s", "%s")) for row in
self.db.album_by_artist(args.artist):

print(p.fmt(row))

if __name__ == '__main__':

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.

6

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 of en want to nd 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 of en 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 of en what you get to see is its

nal form, not the intermediary steps where the author tries things and re ne

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 e cient 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 o fers 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 le, which is read at startup by the *psql* application. As you've already read in the PostgreSQL documentation for *psql*, we have three di ferent 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 speci c 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 a fecting 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 les. When unset or set to o f, 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 ight, so you know you need to nish the transaction. More importantly, when you want to type in anything that

will have a side e fect on your database (modifying the data set or the database

schema), then without the star you know you need to rst type in BEGIN.

Let's see an example output with *ON_ERROR_ROLLBACK* set to o f. 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
```

block

f1db!# rollback;

ROLLBACK

We have an error in our transaction, and we notice that the star prompt is now a

ag. The SQL transaction is marked invalid, and the only thing PostgreSQL will

now accept from us is to nish 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 af er 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 rst case, the idea is that you have plenty of com-

mands 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 com-

mands: 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 le 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 identi ers 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
order by position
limit :n;
```

When running *psql* for reports, it might be good to have a speci c 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 op-

tions.

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 noti cations 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 queri* 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 nd 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 documen-

tation of all the commands available in *psql*, so go read the whole manual page to nd 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 le, 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 lo-

cally.

7

SQL is Code

The rst 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, ltering 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 e ciency is-

sues. Then in the previous section, we approached a good way to have your SQL

queries as .sql les in your code base.

Now that SQL is actually code in your application’s source tree, we need to ap-

ply the same methodology that you’re used to: set a minimum level of expected

quality thanks to common indentation rules, code comments, consistent nam-

ing, 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 im-

portant 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 BY** 2; Here we have a run-away query all on the same line, making it more di cult 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 di f:

```
@@ -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 modi ed 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 lef 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 con-
sistent. 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 condi-
tions 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 before when the SQL standard did spec-
ify 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 ve di ferent 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 nd out if the Chinook data set

includes cases of a track being named af er another artist’s, perhaps re ecting 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 af er 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 rst 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 pre x 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:

```
-- artists names used as track names by other artists
select artist.name as artist,
-- "inspired" is the other artist
inspired.name as inspired,
album.title as album,
track.name as track
```

```
from
artist
/*
* Here we join the artist name on the track name,
* which is not our usual kind of join and thus
* we don't use the using() syntax. For
* consistency and clarity of the query, we use
* the "on" join condition syntax through the
* whole query.
*/
```

```
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;
```

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 di cult 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 nding it easier when we have to debug it in pro-

duction, should we have to. Given the PostgreSQL's *application_name* facility and a proper use of SQL les 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:

1

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 di
ferent

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 le containing your tests with *psql*
2. Capture its output to a text le 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 di ference 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 le 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 les, wherein the setup step builds a database model and lls 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 examples, 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 ngertips, 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

9

(0.06 usr

0.02 sys +

0.08 cusr

0.07 csys =

0.23 CPU**)**

10

Result: PASS

You might also nd 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. Typi-

cally you would run the regression tests each time a query is changed.

The RegreSQL tool implements that idea. It nds SQL les 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 x the problem: either change the expected output (with regresql update) or x the *re*-

gresql/plans/src/sql/artist.yaml le.

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 nding which part of the

code is sending a speci c 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.

8

Indexing Strategy

Coming up with an *Indexing Strate* is an important step in terms of mastering your PostgreSQL database. It means that you are in a position to make an 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 nd 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 of en thought of as a data modeling activity. When using Post-

greSQL, 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 strate is indeed a data modeling activity.

In all other cases, the *indexing strate*

is meant to enable methods for faster ac-

cess 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 strate* 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 per-

forming updates on the same data rows, providing transaction iso-

lation for each database session. MVCC, by eschewing the locking methodologies of traditional database systems, minimizes lock con-

tention 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 con icting 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- ight 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 de ned

in terms of a *constraint*.

Indexing for Queries

PostgreSQL automatically creates only those indexes that are needed for the sys-

tem to behave correctly. Any and all other indexes are to be de ned by the appli-

cation 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 a 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 de ne a global *indexing strate* . Unless you have in nite IO bandwidth and storage capacity, it is not feasible to index everything in your

database.

Choosing Queries to Optimize

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 lack-ing 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 inter-

esting 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 di ferent algorithm

that is best suited to di ferent types of queries. By default, the CRE-

ATE INDEX command creates B-tree indexes, which t the most

common situations.

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

• *B-Tree*, or balanced tree

Balanced indexes are the most common used, by a long shot, because they

are very e cient and provide an algorithm that applies to most cases. Post-

greSQL 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 theoret-

ical background in the source code le:

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 be-

hind 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 *rang* 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 imple-
mentation 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 di ferent densities.

• *GIN*, or generalized inverted index

GIN is designed for handling cases where the items to be indexed are com-
posite 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 docu-
ments containing speci c 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
e ciently handle queries that test for the presence of speci c 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 table. Like GiST, SP GiST and GIN, BRIN can support many di ferent 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 lters

A Bloom lter is a space-e cient 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 sup-

port 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 lter 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 rst *create extension bloom*.

Both *Bloom* indexes and *BRIN* indexes are mostly useful when covering mut-liple 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 strate* . Not every

query needs to be that fast, and the requirements are mostly user de ned. 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 Post-

greSQL restart, because it needs to be registered in shared_preload_libraries

— then it’s possible to have a list of the most common queries in terms of num-

ber 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 appli-

cation. 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. Choos-

ing 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 expe-

rience to master, but this section attempts to cover the basics.

Here’s a very rough guide to using *explain* for xing query performances:

• use the spelling below when using *explain* to understand run time charac-

teristics 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 one included in pgAdmin.

• First check for row count di ferences in between the *estimated* and the *effective* numbers.

Good statistics are critical to the PostgreSQL query planner, and the col-

lected statistics need to be reasonnably up to date. When there’s a huge

di ference in between estimated and e fective row counts (several orders

of magnitude, a thousand times o f or more), check to see if tables are ana-

lyzed 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 *index 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 sum-

mary in a web browser is far too common.

The rst 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 rst query only to run an extra query for each row retrieved.

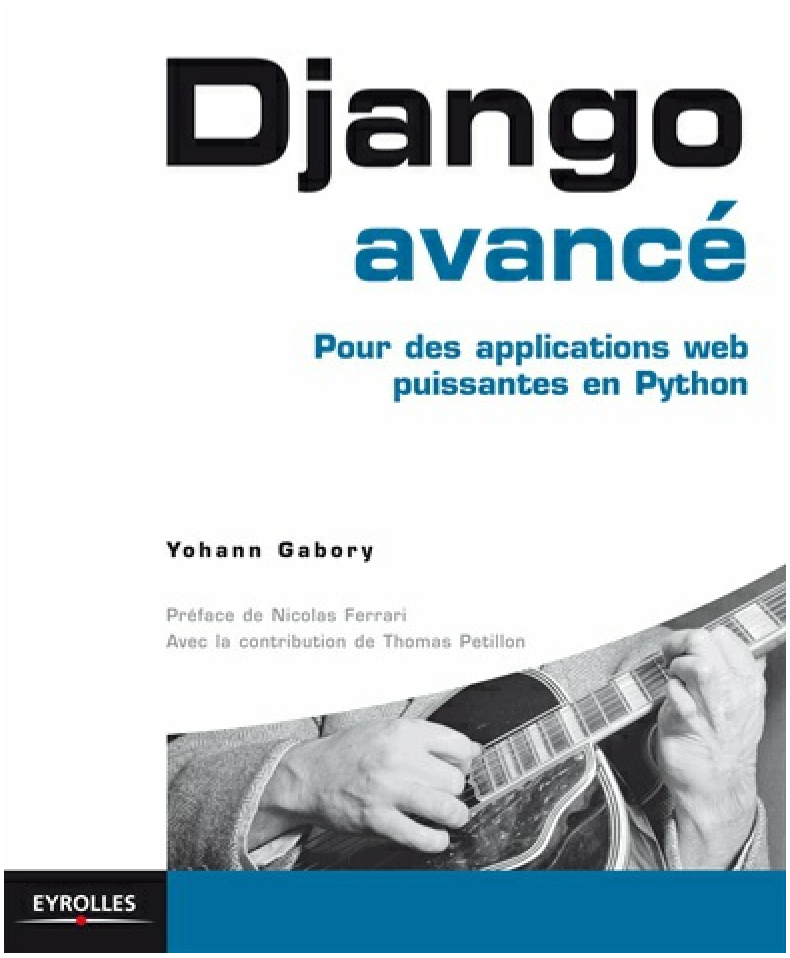

Django
avancé
Pour des applications web
puissantes en Python
Yohann Gabory
Préface de Nicolas Ferrari
Avec la contribution de Thomas Petillon
EYROLLES

Figure 8.1: Advanced Django

9

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?

Consistency and con dence

Data

what a web application reli 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

th

: if you are ready to lose some data then your data h

no

value. If your data h no value then there a big chance that your

app h 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

confidence. Your

users must trust you when you say, "I have saved your data". They

must trust you when you say, "Your data

not corrupted".

So what

the feature I first expect?

Don’t mess up my database with invalid or corrupted data. Ensure

that when my database says something

saved, it really .

Code in SQL

Of course, th means that each time the coherence of my database

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 — proce-

dure, tri ers, check_constraints — those are the most basic featur

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 do not seem very relevant will

be mandatory or something else I tough w mandatory not after

all.

So I need my database to be flexible enough to let me easily change *what*

mandatory and what

not.

Th

point

the main reason some developers fly to NoSQL

databas . Because they see the schemaless *options*

a way to not

carefully specify their database schema.

At first sight th can seem like a good idea. In fact, th

a ter-

rible 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 consis-

tent" records.

I will not talk about writing consistency and relational checks in

code because it reminds me of nightmar called race-conditions *and*

Heisenbugs .

What I really expect from my RDBMS

to let me begin schema-

less 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 di culties to be aware of when using

PostgreSQL?

Django lets you use a lot of different databas . You can use SQLite, MariaDB, PostgreSQL and some others. Of course, you can expect from some databas availability, consistency, isolation, and dura-bility. Th allows you to make decent applications. But there always a time where you need more. Especially some database type that could match Python type. Think about list, dictionary, rang , timestamp, timezone, date and datetime.

All of th (and more) can be found in PostgreSQL. Th 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 com 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 featur . It’s also a really warm place to code with confidence.

Because Django h a migration management system that can handle pure SQL I can write advanced SQL functions and tri ers directly in my code. Those functions can use the most advanced featur 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, a regation functions and more directly in your Django code. And those function you write are plain SQL.

Th also means that version after version your RDBMS choice 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 th

a tough question. Some will say ORM sucks. Some

others says mixing SQL and Python code in your application 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

not sustainable in the long run.

When your queri are so simple you can express them with your

ORM why not use it? It will generate a SQL query

good

anybody could write. It will hydrate a Django object you can use

right away, in a breeze.

Think about:

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

Th

equivalent to:

```
select mymodel.id, mymodel.other_field, ...
from mymodel
```

3

where id=1;

Do you think you could write better SQL?

ORM can manage all of your SQL needs. There also some advice

to avoid the N+1 dilemma. The a regation system reli on SQL

and

fairly decent.

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

The rule of thumb for me

to never forget what your ORM

meant for: translate SQL records into Python objects.

If you think it can handle anything more, like avoiding writing

SQL, managing index etc… you are wrong.

The main Django ORM philosophy

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 th

1

MyModel.objects.filter(...).query.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 y , 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

a grown-up I know two things. Santa and RDBMS

agnosticism do not really exist.

What

true

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 pro-

duction leads only to one thing: you will use the featur of SQLite

everywhere and you will not be able to use the PostgreSQL specific

featur .

The only way to be purely agnostic

to use only the featur all the

proposed RDMS provid . But think again. Do you want to drive *your race car like a tractor?*

Part IV

SQL Toolbox

In this chapter, we are going to add to our pro ciency 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 di ferent

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, rst write down a single sen-

tence —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 e fective techniques in writing such a sentence is talking out loud,

because apparently writing and speaking come from di ferent 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.

Af er having dealt with the basics of the language, where means basic really fun-

damentals, this chapter spends time on more advanced SQL concepts and Post-

greSQL along with how you can bene t from them when writing your applica-

tions, making you a more e fective developer.

Get Some Data

To be able to play with SQL queries, we rst 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 le for MySQL only. Once you

have a local copy, we use pgloader to get the data set in PostgreSQL: 1

```
$ createdb f1db

2

$ 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:

1

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 bene t 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 di fers 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. Af er that, the RDBMS of your choice is going to gure 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 tech-

nology 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 exe-

cuted.

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.

12

Queries, DML, DDL, TCL, DCL

SQL means *structured query language* and is composed of several areas, and each of them has a speci c 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 de ne 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 statements *grant* and *revoke*.

• Next we have PostgreSQL maintenance commands such as *vacuum*, *ana-*

lyze, *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*, *valu* 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 e forts in this section.

13

Select, From, Where

Anatomy of a Select Statement

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

```
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 speci ed 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 al-

lows us to use this alternative, which is even more practical:

1

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 de nitions, then the same

query now has a di ferent result set data structure, and you might have to re ect

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, ltering

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 le 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
```

10 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 *rac* table column list, each column name, and the data types. Of course, it's still pos-sible 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 *rac* table has eight column.

Now pretend we had an *extra* column in our schema de nition 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 ResultSet.
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, be-

cause 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;");

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("extra"));
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 le.

• It is not e cient 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 nd their way in the network and your

application’s memory.

The main point is about being speci c 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:

```
select code,
format('%s %s', forename, surname) as fullname,
forename,
surname
from drivers;
```

And here are the rst three drivers we get:

```
code │
fullname
│ forename │ surname
══════╪════════════════╪══════════╪═════════
HAM
│ Lewis Hamilton │ Lewis
│ Hamilton
HEI
│ Nick Heidfeld
│ Nick
│ Heidfeld
ROS
│ Nico Rosberg
│ Nico
│ Rosberg
(3 rows)
```

Here we are using the format PostgreSQL function, which mimics what is usu-

ally 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
from drivers;
```

In this book, we are going to focus on PostgreSQL rather than standard com-
pliance, because PostgreSQL o fers a lot of useful functions and gems that are
nowhere to be found in the SQL standard, nor in most of the RDBMS compe-
tition.

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 any-
where 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",
extract('week' from date) as week,
extract('day' from
```

```
(date + interval '2 month - 1 day')
)
```

```
as feb,
extract('year' from date) as year,
extract('day' from
(date + interval '2 month - 1 day')
) = 29
as leap
from generate_series(date '2000-01-01',
date '2010-01-01',
interval '1 year')
as t(date);
```

The *generate_seri ()* function returns a set of items, here all the dates of the rst day of the years from the 2000s. For each of them we then compute the day of

the week of this rst day of the year, both in numerical and textual forms, and

then the year number from the date, as de ned 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 de nition, ISO weeks start on Mondays and the rst week of a

year contains January 4 of that year. In other words, the rst Thurs-

day 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 pro-

cessing 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 di ferent sources relate to each other. In the most basic form,

our query is reading a data set from a single table:

1

```
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 nd 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 ll 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:

1

```
\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'

12

+ :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 lef of the

operator, and to ll-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 lter out races from which we don't have a result yet, and we are still interested in 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 lter for the query: when the lter evaluates to true then we keep the row in

the result set and when the lter 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 ltering expressions of our queries.

Some simple rules to remember here:

• In a *where* clause we can combine lters, 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 di ferent beasts.

Be careful about *not in* semantics with *NULL*: the following query re-

turns 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 nish a single race in which they

participated, then we can lter out those who did nish. We know that a driver nished because their *position* is lled in the *results* table: it *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
and date <
:season + interval '1 year'
and not exists
```

```
(
select 1
from results r
where position is not null
and r.driverid = drivers.driverid
and r.resultid = results.resultid
)
group by constructors.name, driverid
order by count(*) desc;
```

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

Remember that a *where* clause is a lter. The *not exists* clause is ltering based on rows that are returned by the subquery. To pass the lter, 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 nish a race so we are going to show only the ten

rst 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 nish a race might be *did not qualify* or *gearbox*, or any one of the 133 di ferent statuses found in the f1db database.

14

Order By, Limit, No Offset

Ordering with Order By

The SQL *ORDER BY* clause is pretty well-known because SQL doesn't guar-

antee 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.

1

```
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 rst, 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 Post-
greSQL. Let’s nd out the ten nearest circuits to Paris, France, which is at longi-
tude 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 de ned:

```
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 ts 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 refer-

ence the query plan of the previous spelling of the query, the dynamic expression

point(lng,lat) looks like this:

1

```
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 de ned only for geometric data types in PostgreSQL. Some extensions such as pg_trgm add to that list so that you may bene t from a kNN index lookup in other situations, such as in queries using

the *like* operator, or even the regular expression operator ~. You'll nd 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 *rac .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)

where
extract('year' from date_trunc('decade', races.date))
= decades.decade

group by decades.decade, drivers.driverid
order by wins desc
limit 3
)
```

```
as winners on true

order by decade asc, wins desc;
```

The query extracts the decade rst, 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* nds the top three winners.

Focusing now on the *winners* subquery, we want to *count* how many times a driver made it to the rst 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 o fers 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:

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

Figure 14.1: No O fset

results, it’s less and less e cient 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 *lap-*

tim 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 rst 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 rst 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 e fort 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
from laptimes
join drivers using(driverid)
where raceid = 972
and row(lap, position) > (1, 3)
order by lap, position
fetch first 3 rows only;
```

And here’s our second page of query results. Af er a rst page nishing at lap 1, position 3 we are happy to nd 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!

15

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* ltering. This lter 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* lter.

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 dif-

ferent groups, and in each group reduce the data set to a single value.

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

```
select extract('year'
from
date_trunc('decade', date))
as decade,
```

count(*)

6

from races

7

group by decade

8

order by decade;

PostgreSQL o fers 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 di ference between each decade is easy to compute thanks to *window func-*

tion, 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)

else format('+%3s',
nbraces
- lag(nbraces, 1)
over(order by decade))

end as evolution
```

```
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 di ference 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 rst, 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 nish

any single race they participated in over their whole career:

```
with counts as
(
select driverid, forename, surname,
count(*) as races,
bool_and(position is null) as never_finished
from drivers
join results using(driverid)
join races using(raceid)
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 nished 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 nd out if some seasons were less

lucky than others on that basis and search for drivers who didn't nish 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 lter. Here we choose

to count all race results where the position is null, which means the driver didn’t

make it to the nish 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:

1

select count(*)

2

```
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 lter the result set to only those groups that meet the *having* ltering 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 nish a race because they didn’t qualify to take part in it. Another quite common reason for not nishing is that the driver

had an accident.

Grouping Sets

A restriction with classic aggregates is that you can only run them through a sin-

gle group de nition 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 af er 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 nally, the last line is the total amount of points for everybody in-

volved 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

di ferent 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 nish the race because of accidents, and

that was even the second reason listed just af er *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 ve 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 e fect 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 partici-

pants 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 rst 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
from tops
join points as champ_driver
on champ_driver.season = tops.season
and champ_driver.constructorid is null
and champ_driver.points = tops.dtops
```

```
join points as champ_constructor
on champ_constructor.season = tops.season
and champ_constructor.driverid is null
and champ_constructor.points = tops.ctops
)
select season,
format('%s %s', drivers.forename, drivers.surname)
as "Driver's Champion",
constructors.name
as "Constructor's champion"
from champs
join drivers using(driverid)
join constructors using(constructorid)
```

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 aggre-

gates 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 di ferent 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 nal 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 rst

row of each set of rows where the given expressions evaluate to

equal. The DISTINCT ON expressions are interpreted using the same rules as for ORDER BY (see above). Note that the “ rst row”

of each set is unpredictable unless ORDER BY is used to ensure

that the desired row appears rst.

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 con-

structors, 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)

where raceid = 971
and points = 0
)
;
```

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 di ference 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 le.

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:

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

This returns 0 rows, so the index is reliable *and* the *location* column is lled 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 di ference in SQL is that *null* introduces *three-valued logic*. Where that's very di ferent 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 rst 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 rst 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 th*

rather than *no value*

here. Say you have in A (lef 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

distinct from and

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 speci c 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'); 3

insert into test(f1) **values**(**DEFAULT**),(**NULL**),('foo');

4

ERROR:

null value **in column** "f1" violates not-null **constraint**

5

DETAIL:

Failing **row** contains (2, **null**).

This time the *insert* command fails: accepting the data would violate the constraint we speci ed 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 satis ed. When the join condition is not satis ed, the outer joins

then ll 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 nal 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*

null:

```
a │ b
═══╪═══
¤ │ t
¤ │ f
¤ │ ¤
(3 rows)
```

17

Understanding Window Functions

There was SQL before window functions and there is SQL af er *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

a regate 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 rst 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 rst thing to understand here is that *frame*.

The array_agg() function is an *a regate* function that builds an array. Let’s use this tool to understand *window fram* :

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

The *array_a ()* 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 de nition over (order by x) actually means over (order by x

rows between unbounded preceding and current row):

```
select x,
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 to 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
```

```
5
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 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 de ne when using the clause PARTITION BY. It al-

lows de ning as *peer rows* those rows that share a common property with the

current row, and the property is de ned 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 rst), 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 rst 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 af er the *where* clause, so you only get to see rows from the available result set of the query.

Available Window Functions

Any and all *a regate* 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 a regate*. 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 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 nish the race. In SQL terms we also

see that we can have two di ferent 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 speci ed 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 af er 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 out-

put 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 intro-

ducing the *from* clause and some join operations. In this section we are going to expand this on this part and look speci cally at what a relation is.

As usual, the PostgreSQL documentation provides us with some enlightenment

(here in its section entitled the FROM Clause:

A table reference can be a table name (possibly schema-quali ed),

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 Carte-

sian 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 nally 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 prop-

erties, that is to say a set of elements all from the same composite data type. The

SQL standard didn't go as far as de ning 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 de ned 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 de ned 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 de ned in advance in *create table* or *create type* statements, or de ned on the y 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 e fectively de ning a relation data type. At query run time, this relation is lled 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, with 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 nally construct the result set, i.e. a relation.

The PostgreSQL optimizer will then re-arrange the computations needed so they’re as e cient 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 speci ed 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 nish 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 nish 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 di ferent 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 satis ed.

The relation of which you want to keep all the rows is pointed to in the

name of the outer join, so it’s written on the lef -hand side in a *left join*

and on the right-hand side in a *right join*.

When the *join condition* is not satis ed, it means you keep some known

data and must ll 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 de nition, known at query planning time.

19

An Interview with Markus Winand

Markus Winand is the author of the very famous book “SQL Performance ex-

plained” 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 19.1: Use The Index, Luke!

Developers of en 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

that it

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

different because it simply

defin the result. Th becom most obvio

in the select clause,

which literally defin the columns of the result. Most of the other

main claus describe which rows should be present in the result. It

important to understand that the author of an SQL statement

do 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

to stop

thinking in imperative terms. One recurring example I've seen in *the field*

how people imagine that joins work and more specifi-

cally, which index can help in improving join performance. Peo-

ple constantly try to apply their knowledge about algorithms to SQL

statements, without knowing which algorithm the database actually

us . Th caus a lot of problems, confusion and frustration.

First, always foc

on writing a clear statement to describe each col-

umn and row of the desired result. If needed, you can take care of

performance afterwards. Th however, requir some understand-

ing 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

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 th job correctly,*

you properly phrased it,

programmers should at least know what their SQL database could

do. Once you remember that SQL can do a regations without

group by—e.g. for running totals, moving averag , 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 lin of SQL can replace dozens of lin of an

imperative program. Most of the time, the SQL solution

more

correct and even faster. In the vein 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 di ferent 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. Th

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 defin a

meaningful behavior, even for the most obscure corner cas . Vendor

extensions have a tendency to foc

on the main cas . For corner

cas , they might behave in surprising and inconsistent ways — just

because nobody thought about that during specification.

Sometim , I cannot solve a problem with standard SQL — at least

not in a sufficiently elegant and efficient way. That

more often

because the database at hand doesn't support the standard featur

that I'd like to use for th problem. However, sometim the stan-

dard just doesn't provide the required functionality. In either case

I'm also happy to use a vendor extension. For me, th

really just

my personal order of preference for solving a problem — it *not a*

limitation in any way.

When it com 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 do 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

not only about the

code — it

also about the people. I’d say it

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 involv less friction. Even from the personal

viewpoint of a single developer, it h a big benefit: if you are used to

writing standard SQL then the chanc increase that you can write

SQL that works on many databas . Th mak you more valuable

in the job market.

However, there

one big exception and that’s DDL – i.e. create

statements. For DDL, I don’t even aim for portability in the first

place. Th

pointless and too restricting. If you need to create

tabl , views, index , and the like for different databas , it better

to just maintain a separate schema definition for each of them.

How do you see PostgreSQL in the RDBMS o fering?

PostgreSQL

in a very strong position. I keep on saying that from

a developer’s perspective, PostgreSQL’s feature set

closer to that of

a commercial database than to that of the open-source competitors

such

MySQL/MariaDB.

I particularly like the rich standard SQL support PostgreSQL h :

that means simple things like the fully featured valu clause, but *also with [recursive], over, lateral and arrays.*

Reading the Wikipedia article on relations in databases article, we nd the following:

In relational database theory, a relation, as originally de ned 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 de nition notwith-

standing, and contrary to the usual de nition in mathematics, there

is no ordering to the elements of the tuples of a relation.[2][3] In-

stead, 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 de nition:

they have the same list of attributes, and each attribute is of a speci c 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.

20

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 le systems.

On this topic, it might be interesting to realize how Lisp introduced *print read-*

ably. In Lisp rather than working with a compiler and then running static binary les, you work with an interactive *REPL* where the *reader* and the *printer* are fully speci ed 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 e fect. Speci cally, printing any object O1 produces a printed

representation that, when seen by the Lisp reader while the standard

readtable is in e fect, will produce an object O2 that is similar to O1.

In the following example code, we de ne a structure with *slots* of di ferent types: string, oat, and integer. Then we create an instance of that structure, with speci c 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)

2

#S(READABLY::FOO :NAME "bar" :X 1.0 :N 2)

3

#S(READABLY::FOO :NAME "bar" :X 1.0 :N 2)
```

The rst instance is created in the application code from literal strings and num-

bers, and the second instance has been created by the reader from a string, which

could have been read from a le 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 mem-

ory isn't a problem for which you need a database system.

21

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 several 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-o ce application, the public front of the application,

the accounting reports, nancial reporting, and maybe some more parts such as

salespeople back-o ce 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 of en

the case that di ferent technical stacks are going to be used for di ferent 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 *schemaless* with the relational modeling and go more deeply into this topic. In order to be

able to compare those very di ferent 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 de nition for *relation* mentions *attribute valu* 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 de nition page: *The term originated*

an abstraction of the sequence: single, dou-

ble, triple, quadruple, quintuple, sextuple, septuple, octuple, …,

n-tuple, …, where the prefix are taken from the Latin nam of

the numerals.

So by de nition, 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 rst business constraint enforced here is dealing with proper data. For in-
stance, the *timestamp* data type in PostgreSQL implements the Gregorian Cal-
endar, in which there’s no year zero, or month zero, or day zero. While other sys-
tems might accept “timestamp formatted” text as an attribute value, PostgreSQL
actually checks that the value makes sense within the Gregorian Calendar:

```
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 Febru-
ary 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:

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 lef 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 = '='
and oprleft::regtype::text ~ 'int|time|text|circle|ip'
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 nd 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

• Speci c implementations of existing functions for the new data type

• Operator speci c 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 go 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 under-standing 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 com-pletely 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 under-
stand 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$)'
```

7

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 di ferent

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 speci c

aggregates for them:

1

```
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 *a regate* it silently bypasses *null* by default, so in our expression of *bool_and(position = 1)* we will lter F1 drivers who won all the races they nished in a speci c 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 nished *and* won every race they entered in a season we need to then write *having bool_and(position not distinct*

from 1)

true, and then the result set only contains those drivers who partici-

pated 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 to test against literal *true*, *false* or *null* rather than =

• Remember to use the

distinct from and

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 ei-

ther 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 Je f Davis.

Character and Text

PostgreSQL knows how to deal with characters and text, and it implements sev-

eral 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 nd them all in the string functions and operators documentation chapter — with functions such as *overlay()*, *substring()*, *position()* or *trim()*. Or aggregates such as *string_a ()*. 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

nd 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 speci ed 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:

1

\pset format wrapped

2

\pset **columns** 70

3

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 *them* column contains several categories for a single entry, separated with a comma. Within that comma separated list, we nd another

classi cation, this time separated with a greater than sign, which looks like a hi-

erarchical 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 embed-

ded in the *them* column?

With PostgreSQL, this is easy enough to achieve. First, we are going to split the

them column using a regular expression:

```
select id, regexp_split_to_table(themes, ',')
```

```
from opendata.archives_planete
```

```
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 cate-

gories 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 af er you’ve imported it into PostgreSQL

makes the di ference 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
```

```sql
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 *categori* table containing each entry we saw in the rollup query only once, as a catalog, and an association table

in between the main *opendata.archiv _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 queri* , *ranking*, *highlighting*, *pluggable parsers*, *dictionari* and *stemmers*, *synonyms*, and *thesaur* . Additionally, it's possible to con gure 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 pos-

sibly 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 integra-

tion for the page size here, so you don't get to see the *Access privileg* 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 but 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 y be-

tween 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 di ferent languages and scripts, all encoded in *UTF8*:

```
language

│

hello
══════════════════════════════╪════════════════

Czech (čeština)

│ Dobrý den

Danish (dansk)

│ Hej / Goddag / Halløj

Dutch (Nederlands)
```

│ Hallo / Dag

English /ˈɪŋɡlɪʃ/

│ 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)

│ Οὖλέ 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)

│ Здра

вствуйте!

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 oating point numbers with 6 decimal digits precision

• *double precision*, 64 bits oating 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 resultset 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 lef to gure 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 di ferent variants when its lef operand is

a *bigint* value:

1

select oprname,

2

oprcode::regproc,

3

oprleft::regtype,

4

oprright::regtype,

5

oprresult::regtype

6

from pg_operator

7

where oprname = '='

8

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:

```
1

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 oating

point numbers. Please take some time to read What Every Programmer Should

Know About Floating-Point Arithmetic before considering any serious use of oating 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 di ferent 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 typ* : 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')
```

4

);

5

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, with-

out 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 ex-

ceed 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 t 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 identi er (*UUID*) is a 128-bit number used to identify information in computer systems. The term globally unique identi er (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 identi ers 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 en-

titled Binary Data Types.

While it’s possible to store large binary data that way, PostgreSQL doesn’t imple-

ment a chunk API and will systematically fetch the whole content when the col-

umn 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 proper-

ties, *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 rst question we need to answer here is about using timestamps with or

without *time zon* from our applications. The answer is simple: always use

timestamps WITH time zon .

A common myth is that storing time zones will certainly add to your storage and

memory footprint. It’s actually not the case:

```
select pg_column_size(timestamp without time zone 'now'),
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 sup-

port. Here’s their whole type de nition 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 speci ed is converted to UTC using the appropriate

o fset 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

o fset 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;
```

```sql
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 Paci c 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 applica-

tion with users in di ferent 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 di ferent 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 di ferent 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 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 coordinat even more than we assume tempo-

ral coordinat , and in the case of time in ordinary communication,

it simply left out entirely. Despite the existence of time zon and

strange daylight saving time regim around the world, most people

are blithely unaware of their own time zone and certainly of how

it relat to standard referenc . Most people are equally unaware

that by choosing a notation that

close to the spoken or written ex-

pression of dat , they make it meaningless to people who may not

share the culture, but can still read the language. It unlikely that

people will change enough to put these issu to rest, so responsible

computer people need to address the issu and resist the otherwise

overpowering urge to abbreviate and drop context.

Several options are available to input timestamp values in PostgreSQL. The eas-

iest 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 o fset 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 *postgr _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'

13

)

14
```

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 speci c 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 times-

tamp 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
```

```
5

limit 1;
```

The column names *ats* and *cts* respectively stand for *author commit timestamp* and *committer commit timestamp*, and the *subject* is the rst 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 com-
mits 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 di ferent, 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 di ference 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'

10

order by ats;

It’s tempting to use the *between* SQL operator, but we would then have to re-

member 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 rst time of the day, which is easier, and well supported by

PostgreSQL.

Also, using explicit bound checks allows us 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 query, 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:

```sql
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 nd 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 ltering 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 of en easier

to share a SQL query than to set up a distributed service API o fering 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 nd 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 le:

```
begin;

drop table if exists access_log;

create table access_log
```

```
(
ip inet,
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 "-" "Mozilla/4.0 (compatible; MSIE 5.5; Windows 98)"

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 elds 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*:

1

```
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. Af er 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 de nitions.

```
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* le into the more interesting format we use here:

```
begin;

create schema if not exists raw;

-- Must be run as a Super User in your database instance
-- create extension if not exists btree_gist;

drop table if exists raw.rates, rates;
```

```
create table raw.rates
(
currency text,
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 rst create a target table for loading the CSV le. The le

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 *rat* table.

The *rat* 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 lls in the *rat* table from the *raw.rat* we'd been importing in the previous step. The query uses the *lead()* window function to implement the speci cation spelled out in English earlier: *an exchange rate relevant from its publication until the next rate 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 rst 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 speci c time to write the following query:

\index{Operators!@}

```
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 e ciently:

```
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 lan-

guage with new capabilities. Speci c 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-de ned 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**,

6

date

date,

7

hour

time,

8

uname

text,

9

nickname

text,

10

bio

text,

11

message

text,

12

favs

bigint,

13

rts

bigint,

14

latitude

double precision,

15

longitude

double precision,

16

country

text,

17

place

text,

18

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 t for PostgreSQL capabili-

ties. The *date* and *hour* elds 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.

```
begin;

create table hashtag
(
id bigint primary key,
date timestamptz,
uname text,
message text,
location point,
hashtags text[]
);

with matches as (
select id,
regexp_matches(message, '(#[^ ,]+)', 'g') as match
from tweet
```

```
),
hashtags as (
select id,
```

```
array_agg(match[1] order by match[1]) as hashtags
from matches
group by id
)
insert into hashtag(id, date, uname, message, location, hashtags)
select id,
date + hour as date,
uname,
message,
point(longitude, latitude),
hashtags
from
hashtags
join tweet using(id);

commit;
```

The PostgreSQL matching function *regexp_match ()* implements what we need

here, with the *g* ag to return every match found and not just the rst tag in a message. Those multiple matches are returned one per row, so we then *group by* tweet id and *array_a*

over them, building our array of tags. Here’s what the

computed data looks like:

```
select id, hashtags
from hashtag
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.

```
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 con rm that our previous index makes sense for looking inside

the *hashtags* array:

```
explain (analyze, verbose, costs off, buffers)
select count(*)
```

```
from hashtag
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 *geonam* 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 ef-cient to work with. Nonetheless, it’s still not so e cient 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 ine cient, so learn to use *unnest()* instead, and lter 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:

```
begin;

create type rate_t as
(
currency text,
validity daterange,
value numeric
);

create table rate of rate_t
(
exclude using gist (currency with =,
validity with &&)
);


```

```
insert into rate(currency, validity, value)

select currency, validity, rate

from rates;


commit;
```

The *rate* table works exactly like the *rat* one that we de ned earlier in this chapter.

```
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 Procedur* API

• Advanced use cases of *array* of *composite* types

XML

The SQL standard includes a SQL/XML which *introduc the predefined data type XML together with constructors, several routin , 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 chap-

ters 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 procedur* 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:

```
create extension plxslt;


CREATE OR REPLACE FUNCTION striptags(xml) RETURNS text
```

```
LANGUAGE xslt
AS $$<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns="http://www.w3.org/1999/xhtml"
```

```
>

<xsl:output method="text" omit-xml-declaration="yes"/>

<xsl:template match="/">
<xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
$$;
```

It can be used like this:

```
create table docs
```

```
(
id serial primary key,
content xml
);

insert into docs(content)
values ('<?xml version="1.0"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>hello</body>
</html>');
```

```
select id, striptags(content)
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 eld. 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 func-

tions and operators, and complete indexing support. The documentation covers

all the details in the chapters entitled JSON Types and JSON Functions and Op-

erators.

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 veri cation that the format is valid *json* input… much like *XML*.

Later, the community realized that the amount of *JSON* processing and ad-

vanced 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 ex-

actly 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 pre-processes 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 draf that is still available for backward compatibility reasons only. Here’s a very quick example showing some di ferences between those two datatypes:

```sql
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 o fer an implementation for the *contains* operator. Switching the content to *jsonb* then:

```
alter table js alter column extra type jsonb;
```

Now we can run the same query again:

```
select * from js where extra @> '2';
```

And we nd out that of course our sample data set of two rows contains the number *2* in the extra *jsonb* eld, 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

elds needed for a part of the data model, and want this data model to be very easily extensible.

In the rst case, using *jsonb* is a great enabler in terms of your application’s capabilities to process the documents it manages, including searching and ltering

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 exible 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-o f sounds interesting from a model design and maintenance perspec-

tive, but is very costly when it comes to daily queries and application develop-

ment: you never really know what you’re going to nd 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-o f 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* eld 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. Af er all, join operations against small reference tables are well supported by the PostgreSQL

SQL engine.

24

PostgreSQL Extensions

The PostgreSQL contrib modules are a collection of additional features for your favorite RDBMS. In particular, you will nd there extra data types such as *hstore*, *ltree*, *earthdistance*, *intarray* or *trigrams*. You should de nitely 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 and production environments from day one.

PostgreSQL extensions are now covered in this second edition of the book.

Figure 24.1: The Postgresql object model manager for PHP

25

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 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.

Considering that you have di ferent layers of code in a web application, for exam-

ple 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

an information chain. At one end there the client that can

run a local application in JavaScript, at the other end, there

the

database. The client calls an application server either synchronously

or asynchronously through an HTTP web service most of the time.

Th data exchange interesting because data are highly denormal-

ized and shaped to fit business needs in the browser. The application

server h the tricky job to store the data and shape them *needed*

by the client. There are several patterns to do that, the most com-

mon

the Model/View/Controller also known

MVC. In th

architecture, the task of dealing with the database

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 imple-

ment 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 o f to

PostgreSQL?

I am essentially dealing with SQL & PHP on a server side. PHP

an object-oriented imperative programming language which means

it good at execution control logic. SQL a set-oriented declarative

programming language and perfect for data computing. Knowing *th , it*

easily understandable that business workflow and data

shaping must be made each in its layer. The tricky question

not

which part of the business logic should be handled by what but how

to mix efficiently these two paradigms (the famo

impedance mis-

match known to ORM users) and th

what the Pomm Model

Manager

good at. Separating business control from data com-

putation also explains why I am reluctant to use database vendor

procedural languag .

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 bene ts and draw-

backs of those constraints when compared to a “schemaless” approach?

The normal form guaranti consistency over time. Th

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 tabl with one primary key per relation. It*

like

tabl were spreadsheets because people foc

on valu . Relational

databas are by far more powerful than that

they emphasize

typ . Tabl are type definitions. With that approach in mind, in-

teractions between tupl can easily be addressed. All typ life cycl

can be modeled th way. Modern relational databas offer a lot of

tools to achieve that, the most powerful being ACID transactions.

Somehow, for a long time, the normal form w

a pain when it

w

to represent extensible data. Most of the time, th data had

no computation on them but they still had to be searchable and

at least … here. The support of unstructured typ like XML or

JSON in relational databas

a huge step forward in focusing on

what's really important. Now, in one field there can be labels with

translation, multiple postal address , media definitions, etc. that *were creating a lot of noise in the database schem*

before. These

are application-oriented structur . It means the database do 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 w about finding a way

to mix SQL & PHP in order to leverage Postgr featur in appli-

cations. Marrying application object oriented with relational not

easy, the most significant step to understand that since SQL us a

projection (the list of fields in a SELECT) to transform the returned

type, entiti had to be flexible objects. They had to be database ig-

norants. Th

the complete opposite of the Active Record design

pattern. Since it not possible to perform SQL queri from entiti

it becom difficult to have nested loops. The philosophy really sim-

ple: 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) entiti . Each entity h*

one or more model class that de-

fine custom queri and a default projection shared by thes queri .

Furthermore, it

very convenient to write SQL queri and use a

placeholder in place of the list of fields of the main SELECT.

Part VI

Data Modeling

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-o fs it involves.

Show me your flowcharts and conceal your tabl , and I shall con-

tinue to be mystified. Show me your tabl , and I won’t usually

need your flowcharts; they’ll be obvio .

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-

o fs, some SQL queries turn out to be really di cult 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 nor-

mal 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 speci c 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 di ference between the application’s design of its internal

state (object-oriented or not) and the database model, though:

• The application implements work ows, 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 speci c to

a set of *user stori* making up a solid part of the whole product.

For example, in a marketplace application, the user publication system is dedi-

cated 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 work ow from systemic consis-

tency, 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 di ferent from object modeling. There are reliable

snapshots of a constantly evolving world on the one side, and transient in-ights

work ows on the other side.

27

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 under-

stand the model when rst 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 compat-

ibility to existing use cases is much harder and time consuming than writing the

rst version. And the rst 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

les to handle your schema, of course.

Some visual tools allow one to connect to an existing database schema and pre-

pare 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

les, 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.

```
1

create database sandbox;
```

Now you have a place where to try things out without disturbing existing appli-

cation 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 re ne it as you go, trying things
out. Here’s a simple and e fective trick to enable that: write your schema as a SQL
script with explicit transaction control, and nish it with your testing queries and
a *rollback.*

In the following example, we iterate over the de nition 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 rst version of our schema and populates

it with random data following the speci cations 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.

```
begin;


create schema if not exists sandbox;


create table sandbox.category
(
id serial primary key,
name text not null
);


insert into sandbox.category(name)
values ('sport'),('news'),('box office'),('music');


create table sandbox.article
```

```
(
  id bigserial primary key,
  category integer references sandbox.category(id),
  title text not null,
  content text
);

create table sandbox.comment
(
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 varia-tions 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 rst 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 de nition 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 ine cient

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 rst three comments on each article.

That’s when we realize our previous schema design misses publication times-

tamps for articles and comments. We need to add this information to our draf

model. As it is all a draf 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:

```
begin;

create schema if not exists sandbox;

create table sandbox.category
(
  id      serial primary key,
  name    text not null
);

```

```
insert into sandbox.category(name)
values ('sport'),('news'),('box office'),('music');

create table sandbox.article
(
id bigserial primary key,
category integer references sandbox.category(id),
pubdate timestamptz,
title
```

```
text not null,
content text
);

create table sandbox.comment
(
id bigserial primary key,
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 func-
tion that’s not provided by default in PostgreSQL, and here’s its de 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 rst query of the product's *MVP*, as speci ed before, on this schema draf 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 rst 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,
```

```sql
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 rst 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 ts 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 nd more on caching later in this chapter.

Our schema is a good rst 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 f y thousands of comments.

• The schema allows an easy implementation of work ows for editing cate-

gories, articles, and comments.

This draf schema is a SQL le, so it’s easy to check it into your versioning sys-

tem, 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

the process of organizing the columns

(attribut) and tabl (relations) of a relational database to reduce

data redundancy and improve data integrity. Normalization

also the process of simplifying the design of a database so that it

achiev the optimal structure. It w

first proposed by Edgar F.

Codd,

an integral part of a relational model.

Data Structures and Algorithms

Af er having done all those SQL queries and reviewed *join* operations, *grouping* operations, ltering 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 t in memory and PostgreSQL has to spill to disk. The planner and optimiser in Post-

greSQL 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 dominat . If you've chosen the right data structur

and organized things well, the algorithms will almost always be self-

evident. Data structur , not algorithms, are central to program-

ming. (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 interfac .

2. *Rule of Clarity*

Clarity

better than cleverness.

3. *Rule of Composition*

Design programs to be connected to other programs.

4. *Rule of Separation*

Separate policy from mechanism; separate interfac from engin .

5. *Rule of Simplicity*

Design for simplicity; add complexity only where you must.

6. *Rule of Parsimony*

Write a big program only when it

clear by demonstration that nothing

else will do.

7. *Rule of Transparency*

Design for visibility to make inspection and debu ing easier.

8. *Rule of Robustness*

Robustness

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 h nothing surprising to say, it should say nothing.

12. *Rule of Repair*

When you must fail, fail noisily and

soon

possible.

13. *Rule of Economy*

Programmer time

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 o fer 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 re ected directly in

the choice of correct data types with advanced behavior and processing function

availability.

To summarize all those rules and the di ferent levels for normal forms, I believe

that you need to express your *intentions* rst. 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

o fers a practical guide to them. In this quick introduction to database normal-

ization, we include the de nition 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 (eld) 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 de nition

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 de nition 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 of en reach for *BCNF* or *4NF* ; going all the way to the *DKNF* design is only seen in speci c cases.

Database Anomalies

Failure to normalize your model may cause *database anomali* . Quoting the

wikipedia article again:

When an attempt

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 updat

to the relation may result in logical inconsistenci . For example, each

record in an “Employe ’ Skills” relation might contain an Employee ID,

Employee Address, and Skill; th

a change of address for a particular

employee may need to be applied to multiple records (one for each skill). If

the update only partially successful — the employee's address updated

on some records but not others — then the relation left in an inconsistent

state. Specifically, the relation provid conflicting answers to the question

of what th particular employee's address . Th phenomenon

known

an update anomaly.

• *Insertion anomaly*

There are circumstanc in which certain facts cannot be recorded at all.

For example, each record in a "Faculty and Their Cours " 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

teach at least one course, but we cannot record a newly hired faculty

member who h

not yet been assigned to teach any cours , except by

setting the Course Code to null. Th phenomenon known

an insertion

anomaly.

• *Deletion anomaly*

Under certain circumstanc , deletion of data representing certain facts

necessitat deletion of data representing completely different facts. The

“Faculty and Their Cours ” relation described in the previo

example

suffers from th type of anomaly, for if a faculty member temporarily ceas

to be assigned to any cours , 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. Th phenomenon

known

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-o
fs are possible with the normalization process, as in the following example.

Modeling an Address Field

Modeling an address eld 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
con-

necting 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 nding the most e cient route

in terms of fuel consumption, time spent and how many packages you can deliver

in a single shif .

Then an address eld looks quite di ferent 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 speci cs of your business.

• Also, if we run a business that delivers to the door (and for example as-sembles furniture, or connects electricity or internet to people homes), we need per house and per-apartment information for each number in a spe-ci c 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 ve following tables, written in pseudo SQL (meaning that this code won’t actually run):

1

```
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 ad-
dress 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 pro-
cess for all those countries, regions and cities where your business operates.

Bor-

ders 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 b or *4 ter*. So even the number information isn't an *integer* eld…

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 rst and

second normal forms. The rst rule to follow to reach rst 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 de nition 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. Compar-

ing *null* values in SQL is a complex matter — as seen in Three-Valued Logic, and rather than argue if the no-duplicate rule applies to *null = null* (which is *null*) or to *null*

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 de ned on an automatically generated column, which is arguably not really part of the data set, then we open the gates for violation of

the rst normal form.

Earlier in this chapter, we draf ed 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'),

3

(2, now(), 'Hot from the Press')

4

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 di ferent — and it is an arti cial 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 publica-

tion date. I don't suppose this is an acceptable situation for the business rules.

In term of database modeling, the arti cially 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 x our schema to prevent duplicate entries:

1

```
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 di ferent 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 di fer-
ent publication dates. It might be needed, af er all, as we know that history of en
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-o f where you have the generated summary key bene ts 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 di ferent 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 enforces 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 unspeci ed 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* de nition or a *check* constraint. The domain de nition applies a *check* constraint to a data type de -

nition. 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 de ne a new data domain as per the PostgreSQL documenta-

tion 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 de nition 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 de ne *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 &&)
);
```

29

Practical Use Case: Geonames

The GeoNam geographical database covers all countri and contains over eleven million place nam that are available for down-

load free of charge.

The website o fers online querying and all the data is made available to download

and use. As is of en 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,
```

13

feature_class

text,

14

feature_code

text,

15

country_code

text,

16

cc2

text,

17

admin1_code

text,

18

admin2_code

text,

19

admin3_code

text,

20

admin4_code

text,

21

population

bigint,

22

elevation

bigint,

23

dem

bigint,

24

timezone

text,

25

modification

date

```
);

create table raw.country
(
iso text,
iso3 text,
isocode integer,
fips text,
```

name

text,

35

capital

text,

36

area

double precision,

37

population

bigint,

38

continent

text,

39

tld

text,

40

currency_code

text,

41

currency_name

text,

42

phone

text,

43

postal_code_format

text,

44

postal_code_regex

text,

45

languages

text,

46

geonameid

bigint,

47

neighbours

```
text,
fips_equiv
text
);


\copy raw.country from 'countryInfoData.txt' with csv delimiter E'\t'


create table raw.feature
(
code
text,
description text,

comment text
);


\copy raw.feature from 'featureCodes_en.txt' with csv delimiter E'\t'


create table raw.admin1
(
code text,
name text,
ascii_name text,
   geonameid   bigint
);

\copy raw.admin1 from 'admin1CodesASCII.txt' with csv delimiter E'\t'

create table raw.admin2
 (
   code        text,
   name        text,
ascii_name text,
geonameid bigint
);

\copy raw.admin2 from 'admin2Codes.txt' with csv delimiter E'\t'

commit;
```

Once we have loaded the raw data from the published les at [http://download.geonames.org/export/dump/,](http://download.geonames.org/export/dump/) we can normalize the content and begin to use the data.

You might notice that the SQL le above is missing the \copy command for the *raw.geonam* table. That’s because *copy* failed to load the le 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 le, 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 de nition 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 man-

aged.

The *raw.geonam* table uses several reference data that *GeoNam* provide as separate downloads. We then need to begin with xing the reference data used

in the model.

Features

The *GeoNam* 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 *featureCod _en.txt* le. Some of the information we need is only available in a text form and has to be reported manually.

```
begin;


create schema if not exists geoname;


create table geoname.class

```

```
(
class char(1) not null primary key,
description text
);


insert into geoname.class (class, description)
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 le 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 le 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 *languag* 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 de-

pendent on the country.

A good way to check for dependencies on the key attributes is with the following

type of query:

1

select currency_code, currency_name, count(*)

2

from raw.country

3

group by currency_code, currency_name

4

order by count **desc**

5

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 for-

mats of countries and focus on some information only. That gives us the follow-

ing normalization process:

1

begin;

2

```
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,
neighbour.tld

from geoname.neighbour as border

join geoname.country as country
on border.isocode = country.isocode

join geoname.country as neighbour
on border.neighbour = neighbour.isocode
```

```
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 *GeoNam* website then o fers an interesting geographical breakdown in the *country_code*, *admin1_code* and *admin2_code*.

```
select geonameid, name, admin1_code, admin2_code
from raw.geonames
where country_code = 'FR'
limit 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 *GeoNam*

website provides

les *admin1Cod ASCII.txt* and *ad-*

min2Cod .txt for us to use to normalize our data. Those les again use admin

codes spelled as *AD.06* and *AF.01.1125426* where the *raw.geonam* table uses them as separate elds. 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:

```
begin;

create schema if not exists geoname;

create table geoname.region
(

isocode integer not null references geoname.country(isocode),
regcode text not null,
name text,
geonameid bigint,

primary key(isocode, regcode)
);

insert into geoname.region
```

```
with admin as
(
select regexp_split_to_array(code, '[.]') as code,
name,
geonameid
from raw.admin1
)
select country.isocode as isocode,
code[2] as regcode,
admin.name,
admin.geonameid
from admin
join geoname.country
on country.iso = code[1];

create table geoname.district
(
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
and d.regcode = r.admin1_code
and d.discode = r.admin2_code
where country_code = 'FR'
limit 5
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:

```
select count(*) as all,
count(*) filter(where country_code is null) as no_country,
count(*) filter(where admin1_code is null) as no_region,
count(*) filter(where admin2_code is null) as no_district,
count(*) filter(where feature_class is null) as no_class,
count(*) filter(where feature_code is null) as no_feat
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 elds 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
on continent.code = country.continent
group by continent.name
order by continent.name;
```

We can see that the *GeoNam* data is highly skewed towards Asia, North Amer-
ica, 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 de nition:

```
1

create index on geoname.geoname using gist(location);
```

Such an index is useful when searching for a speci c 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 *GeoNam* data thanks to the following *lateral left join* lookup in which we implement a *kNN* search with order by ... <-> ... limit k clause:

```
select id,
round((hashtag.location <-> geoname.location)::numeric, 3) as dist,
country.iso,
region.name as region,
district.name as district
from hashtag
left join lateral
(
select geonameid, isocode, regcode, discode, location
from geoname.geoname
order by location <-> hashtag.location
limit 1
)
```

```
as geoname
on true
left join geoname.country using(isocode)
left join geoname.region using(isocode, regcode)
left join geoname.district using(isocode, regcode, discode)
order by id
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 *GeoNam* 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:

```
begin;

create schema if not exists sample;

drop table if exists sample.geonames;
```

```
create table sample.geonames
as select /*
* We restrict the “export” to some columns only, so as to
* further reduce the size of the exported file available to
* download with the book.
*/
geonameid,
name,
longitude,
latitude,
feature_class,
feature_code,
country_code,
admin1_code,
admin2_code,
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 di ferent arguments.) These two methods each return a randomly-chosen sample of the table that will contain approximately the speci ed percentage of the table's rows. The BERNOULLI method scans the whole table and selects or ignores individual rows independently with the speci ed probability. The SYSTEM method does block-level sampling with each block having the speci ed chance of being selected; all rows in each selected block are returned. The SYSTEM method is signi cantly faster than the BERNOULLI method when small sampling percentages are speci ed, but it may return a less-random sample of the table as a result of clustering e fects.

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.geonam* table only contains 115,904 rows. Another run of the same query yielded 115,071 instead. Af er 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 speci cations. 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 af er all. So there we go:

```
begin;


create schema if not exists eav;
create table eav.params
(
  entity    text not null,
  parameter text not null,
  value     text not null,

  primary key(entity, parameter)
);

commit;
```

You might have already seen this model or a variation of it in the eld. The model

makes it very easy to add *things* to it, and very di cult to make sense of the accumulated data, or to use them e fectively 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'),

9

('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 di ferent 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* elds 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 nd 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 *re-*

sponse 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'


join eav.params as etime
on etime.entity = ctype.name
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 com-

plex 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 (eld) are of the same kind.

An anti-pattern that fails to comply with those rules means having a multi-valued

eld in a database schema:

```
create table tweet
(
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 nd 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 o fers 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 e cient 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 x the model!

• Usage Statistics per Tag

For the same reasons that implementing search is di cult, 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 di ferent spellings for the tags, so we might want to normalize them in our database. As we keep the message unaltered in a di ferent 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 enormo* *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 im-

pact when debu ing and maintenance are considered. We should

forget about small efficienci , say about 97% of the time: premature

optimization

the root of all evil. Yet we should not pass up our

opportuniti in that critical 3%.

“Structured Programming with Goto Statements”. Computing

Surveys 6:4 (December 1974), pp. 261–301, §1.

Database modeling has a non-trivial impact on query performance and as such is

part of making attempts at upping e ciency. 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 de ned 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, *sequenc* and their usage as a default value for synthetic keys o fer a guarantee against collisions.

UUIDs on the other hand rely on a way to produce random numbers in a 128

bits space that o fers 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.

31

Denormalization

When modeling a database schema for your application or business case, the very

rst 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 work ow oriented queries, of en 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* work ow 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 eld.

Once all of that is done, some di culties may appear, either because the fully

normalized schema is too heavy to deal with at the application level without any

bene ts, or because having a highly normalized schema involves performances

penalties that you’ve measured and cannot tolerate.

Fully normalized schemas of en 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 di cult to sustain

a fully normalized schema, then it might be time to nd trade-o fs. The process

of *denormalization* consists of relaxing the *normalization* rules to reach an acceptable trade-o f in terms of data quality and data maintenance.

As in any trade-o f 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 mod-

eling 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 di ferent 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 of en than not, performance isn’t the

most important feature for your business. Af er a certain threshold, poor per-

formance 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 de-*

pendency rules and *repeat* data at di ferent 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 de-

normalizing.

Denormalization with PostgreSQL

When using PostgreSQL denormalization may happen by choosing to use de-

normalized 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 mul-

tiple copies of the same data with di ferences 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 terms 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, rst 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 of en. 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 of en 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 rst 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 drif ed 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 af er every race and implementing cache in-

validation 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* de nitions 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 speci c 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 di ferent 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 refer-

ence changes and you want to keep a history that, by de nition, 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 ll 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 di ferent 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, of en enough the

application and business needs surrounding the history entries are relaxed com-

pared 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 af er

its date of validity. When doing nancial 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 tri er* 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 han-

dle 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 bene cial.

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 e cient

at times to manage the list of entries as an array attribute in our main table. This

is particularly e fective when the application of en 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-o f 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. Quot-

ing the PostgreSQL documentation:

• There is no facility available to create the matching indexes on all parti-

tions automatically. Indexes must be added to each partition with sepa-

rate 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 refer-

ences 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 in-

dividual 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 de ned on individual partitions, not

the partitioned table.

So when using *partitioning* in PostgreSQL 10, we lose the ability to reach even the rst *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 home-

work: 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* o fer another set of interesting trade-o fs to implement speci c 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 applica-

tion 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 exam-

ples of query rewrite improving durations from minutes to milliseconds are com-

monly achieved, in particular against queries written by ORMs or other naive

toolings.

Second, denormalization is an optimization technique meant to leverage trade-

o fs. Allow me to quote Rob Pike again, as he establishes his rst rule of programming in Notes on Programming in C as the following: *Rule 1. You can't tell where a program going to spend its time. Bottlenecks occur in surprising plac , so don't try to second guess and* *put in a speed hack until you've proven that's where the bottleneck*

.

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.

32

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 Concur-

rency 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* agship term and are very diverse in their capabilities and behavior.

Under the *NoSQL* term, we nd 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, of en 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 o fered by traditional database systems al-

lows some of the *NoSQL* solution to handle more concurrent activity, of en 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 o fers several ways to relax its *ACID* guarantees and it can be compared favorably to most of the *NoSQL* and *NewSQL* o ferings, 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. In-

stead, 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 elds, are hard-coded into

the application code.

A readily available JSON data set is provided at https://mtgjson.com that *Provid Magic: the Gathering* card data in JSON format, using the [CC0](CC0) license.

We can load it easily given this table de nition:

```
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 x 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 nds a JSON document inside another JSON document. Our *GIN* index de nition 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 elds 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 af er 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 *synchrono _commit* to be set di ferently for each concurrent transaction of the system, and to be changed in- ight within a transaction. Af er

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 dif-

ferent *durability* policies in the same application would be to assign a di ferent level of guarantee to di ferent users:

1

create role dbowner **with** login;

```
create role app with login;

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 work ows. Then the *critical, notsomuch* and *dontcare* roles will have the same set of privileges as the *app* role, and maybe host a di ferent 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 *synchrono _commit* setting in- ight, 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 *synchrono _commit* set-

ting.

Sometimes though, even with relaxing the *durability* guarantees, business re-

quirements can’t be met with a single server handling all the write tra c. 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 e forts. 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 nod* 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 indepen-

dent 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 Post-

greSQL instance.

When considering a *scaling out* solution, always rst 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 o fer no global transactions, which means

no isolation from concurrent activity and as a result there is no possibility to

implement a consistent online backup.

33

An interview with Álvaro

Hernández Tortosa

IT entrepreneur, founder of two sof ware development companies (8Kdata, Wiz-zbill). Sof ware architect and developer. Open source consultant and supporter.

Ávaro 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 nds 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 scenar-

ios.

One my own experience with dynamic schema (please let me avoid

the schema-less term, which I think it completely flawed. I prefer

dynamic schema or schema-attached). Since ToroDB replicat 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 ourselv .

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 h

“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*

a best-practice, and they even offer full cours

on schema design on MongoDB! And once you understand th and need to design the schema, you realize you are basically limited to the following options:

a. *Embed 1:1 relationships inside the documents (th*

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 cas for dynamic schema, like very

flat data structur , or

a temporary store for data of very dynamic

properti , which you may normalize later. But it’s not the unicorn

we have been told to believe it .

The second scenario

related to analytics performance. Basically,

NoSQL

not *designed for analytics purpos and perform very*

poorly. We found 1-2 orders of magnitude speedup when performing

the same queri on relational-structured data vs. NoSQL.

Th may sound counterintuitive: after all NoSQL

for “Big

Data”, isn’t it? Well, it could also be explained in a very intuitive

manner: NoSQL data

unstructured data. And unstructured

data,

it name impli ,

unstructured, that , doesn’t have

an a priori structure. It

a bit “chaotic”, unorganized. Data

may be present, absent, or located anywhere.

And what

analytics? Obtaining valuable information from data. But if data

unstructured, every analytic query needs to parse and analyze

every single document present and infer its structure, check if the

query predicate match the document (or even if the keys that

are looking for even exist on th document!) and so forth. Th

represents a significant extra effort that

completely not required

in relational datastor . And hence they are able to perform even

orders of magnitude faster. Queri that take hours in NoSQL

may take just a few seconds in relational. It’s th dramatic. For

more information, feel free to read our blog post and benchmarks

on th

topic: [*https://www.8kdata.com/blog/announcing-torodb-stampede-1-0-beta/.*](https://www.8kdata.com/blog/announcing-torodb-stampede-1-0-beta/)

With ToroDB Stampede it’s now possible to have both MongoDB and Post-

greSQL feature sets on top of the same live data set. How would you compare

the query languages for MongoDB and PostgreSQL?

MongoDB query language h been growing, adding new operators

and functionality and I expect th trend to continue with every re-

lease. However, if you compare it feature-wise with SQL, especially

with PostgreSQL’s very rich feature SQL implementation, it

al-

most night and day. To name a couple of exampl , joins are only

limited in a very limited fashion, and there are no window func-

tions. What

worse

that some query patterns in MongoDB are

not optimized and performance vari 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

another issue. MongoDB’s query language *a JSON*

document, and it soon becom 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

a standard, and even if

there are some minor differenc between implementations and the

standard itself (by the way, PostgreSQL here do a very good job,

following the standard very closely), it h

led to the development,

for many years, of a huge ecosystem of tools and applications that

you can use with the database. There

simply no such ecosystem

for MongoDB, it's just a minor fraction in comparison.

Note: sure, MongoDB h

the proprietary BI Connector, which

theoretically allows you to connect MongoDB to any SQL tool. The *true story that BI Connector performance very poor when com-*

pared to a SQL database, and its SQL compatibility support also

very small. So it just works on some limited number of cas .

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 typ are really rich and flexible. You can very

easily create your own or extend others. JSONB, in particular,

emulat a whole document, and also supports quite advanced in-

dexing (B-tree index on an expression of the JSON document, or

specialized json index that index either the whole document or

paths within it). One very obvio

question

whether jsonb data

type can compete with MongoDB on its own field, dynamic schema.

One the one hand, MongoDB

not only chosen because of

the dynamic schema, but other capabiliti

such

built-in

high-availability (with its own gotch , but after all integrated

into core) and distributed query. On the former, PostgreSQL

cannot compete directly, there

no HA solution in-core, even

though there are several external solutions. As for distributed

queri , more related to the topic being discussed, there

also not

support per se in PostgreSQL (however you may use Cit

Data's

PostgreSQL extension for a distributed data store or Greenplum

for data warehousing capabiliti). 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 purpos of a docu-

ment store, and it's probably better in some are . 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 issu that MongoDB query language

h). But it offers the best of both worlds: you can freely combine

unstructured with structured data. And th

, indeed, a very com-

pelling use-case: the benefits of a normalized, relational schema de-

sign for the core parts of the data, and those that are obvio

and

clear from the beginning; and add

needed jsonb columns for

less structured, more dynamic, changing data, until you can under-

stand its shape and finally migrate to a relational schema. Th

really the best of both worlds and my best recommendation.

Part VII

Data Manipulation and Concurrency Control

In the previous chapters, we saw di ferent ways to fetch exactly the data you're

interested into from the database server. This data that we've been querying us-

ing 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 concur-

rency 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**,

6

date

date,

7

hour

time,

8

uname

text,

9

nickname

text,

10

bio

text,

11

message

text,

12

favs

bigint,

13

rts

bigint,

14

latitude

double precision,

15

longitude

double precision,

16

country

text,

17

place

text,

18

picture

```
text,
followers bigint,
following bigint,
listed bigint,
lang text,
url text
);
```

```
\copy tweet from 'tweets.csv' with csv header delimiter ';'
```

27

28

```
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 *geonam* 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 of en in production.

It’s now time to rewrite our schema, and here’s a rst step:

```
begin;


create schema if not exists tweet;


create table tweet.users
(

```

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,
listed bigint,

unique(uname)
);

create table tweet.message
(
id bigint primary key,
userid bigint references tweet.users(userid),
datetime timestamptz not null,
message text,
favs bigint,
rts bigint,
location point,
lang text,
url
```

```
text
);

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* elds 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:

```
begin;


create schema if not exists tweet;
create table tweet.users
(
userid bigserial primary key,
uname text not null,
nickname text,
bio text,
picture text,
unique(uname)
);
```

```
create table tweet.follower
(
follower bigint not null references tweet.users(userid),
following bigint not null references tweet.users(userid),

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.

35

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 di ferently in each statement though, and it is included in the SQL stan-

dard too.

Insert Into

Given our model of tweets, the rst thing we need are users. Here’s how to create

our rst users:

```
insert into tweet.users (userid, uname, nickname, bio)
values (default, 'Theseus', 'Duke Theseus', 'Duke of Athens.');
```

The SQL standard *valu* 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 *valu* .

Insert Into … Select

The *insert* statement can also use a query as a data source. We could, for instance, ll in our *tweet.follower* table with people that are known to love each other from their *bio* eld; 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 elds, 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 di ferent 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 con ict 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.

```
begin;

update tweet.users
set nickname = 'Robin Goodfellow'
where userid = 17 and uname = 'Puck'
returning users.*;

commit;
```

Here we pick the id 17 from the table af er a manual lookup. The idea is to show

how to update elds 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* eld, 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 nd no matching rows

and a fect 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 rst 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 xing up some data from the console for a

one-o f x. 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* eld.

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* af er *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 *
)
select min(userid), max(userid),
count(*),
array_agg(uname)
```

```
from deleted_rows;
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 *tupl* and *rows* at di ferent times. There's a di ference 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 *tupl* 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 e cient 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 les 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 com-

mands, 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 lif ing in the background. It is then more e cient 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 e cient. 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 tra c to both tables while they run. If you don’t have slow hours or even o f-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 tra c 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 trans-

actions 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 of en 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

o fer *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 de-

clare our intention: the result we want to achieve. The implementation is then

tasked with implementing our command and making it right in every detail, in-

cluding concurrency behavior.

PostgreSQL implementation of the concurrency behavior is dependable and al-

lows 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 isola-

tion 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 le. 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 de ned by the SQL standard and implemented in PostgreSQL:

The SQL standard de nes four levels of transaction isolation. The

most strict is Serializable, which is de ned by the standard in a para-

graph which says that any concurrent execution of a set of Serial-

izable transactions is guaranteed to produce the same e fect as run-

ning them one at a time in some order. The other three levels are de-

ned in terms of phenomena, resulting from interaction between

concurrent transactions, which must not occur at each level. The

standard notes that due to the de nition of Serializable, none of

these phenomena are possible at that level. (This is hardly surpris-

ing – if the e fect 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 transac-

tion.

• Nonrepeatable read

A transaction re-reads data it has previously read and nds that data has

been modi ed 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 nds 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 inconsis-

tent with all possible orderings of running those transactions one at a time.

There are four isolation levels de ned 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 de nition 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 satisfy-

ing mathematical model for implementing *serializable snapshot isolation* in an e cient 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* eld in the *tweet.message* table. Here’s how to make a *retweet* in our model:

```
update tweet.message
set rts = rts + 1
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:

```
begin;

update tweet.message
set rts = rts + 1
where messageid = 1;
returning messageid, rts;

commit;
```

Now, rather than doing this query, we open a *psql* prompt and send in:

```
begin;

update tweet.message
set rts = rts + 1
where messageid = 1
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 rst transaction is not done yet and is working on the row where *messageid = 1*. Until the rst transaction is done, no concurrent activity can take place on this speci c row.

So we go back to the rst 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 rst transac-

tion to nish — commit the rst 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 com-

mit 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 efficienci , say about 97% of the time:*

premature optimization the root of all evil. Yet we should not pass

up our opportuniti in that critical 3%.

Database systems have been designed to handle concurrency so that your appli-

cation's code doesn't have to. One part for the critical 3% is then related to con-

current 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 rst version, retweeting has to wait until all concurrent retweets are done, and the business model wants to sustain as many concur-

rent 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* eld, 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 e fect 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 wait 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 di ference:

```
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 di ference 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* eld of the *tweet.message* row that you target, and the concurrency behavior is going to be ne 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 invalida-

tion 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*

command.

1

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 num-

bers 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 de nition 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 rst 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
```

7

order by messageid;

We can see that I played with the generating some retweets in my local testing,

done mainly over the six rst 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 di ferent 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 di ferent 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,
       rts, favs,
       location, lang, url
  from tweet.message_with_counters;

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 speci c command. We

have e fectively 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: 1

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 minut ago* rather than *yesterday*. When the *refresh materialized view* command runs in less than ve minutes then implementing the policy is a matter of scheduling that command to be executed

every ve minutes, using for example the *cron* Unix task scheduler.

38

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 *tri er*.

A *tri er* allows registering a procedure to be executed at a speci ed 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, main-

tained separately from the PostgreSQL core. You can nd 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 bene t from the PostgreSQL trigger capabili-

ties.

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.

```
begin;

create table twcache.daily_counters
(
day date not null primary key,
rts bigint,
de_rts bigint,
favs bigint,
de_favs bigint
);

create or replace function twcache.tg_update_daily_counters ()
returns trigger
language plpgsql
as $$
declare
begin
```

```
update twcache.daily_counters
set rts = case when NEW.action = 'rt'
then rts + 1
else rts
end,
de_rts = case when NEW.action = 'de-rt'
then de_rts + 1
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;

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 x 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 rst transaction of the day attempts to *update* the daily counters table for this day, but nds no records because it’s the rst one.

2. The rst transaction of the day then *inserts* the rst 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, nds 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 rst transaction of the day attempts to *update* the daily counters table for this day but nds no records because it’s the rst one.

2. The second transaction of the day attempts to *update* the daily counters

table for this day, but nds no records, because the rst one isn’t there yet.

3. The second transaction of the day now proceeds to *insert* the rst value

for the day, because the job wasn’t done yet.

4. The rst transaction of the day then *inserts* the rst value… and fails with a *primary key* con ict 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 con-

currently, 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 x the problem with the previous trigger implementa-

tion, 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 le 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 le 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 e forts 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.

39

Listen and Notify

The PostgreSQL protocol includes a streaming protocol with *COPY* and also

implements asynchronous messages and noti cations. 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 ow 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 ow, 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 noti

-

cation is sent, the daemon program processes it as necessary, possibly updating

our *twcache.counters* table. As we have a single daemon program listening to no-ti cations and updating the cache, we now bypass the concurrency issues.

Before implementing the client application, we can implement the trigger for

noti cation, 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 noti cations. 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 noti cation only. On the other hand, no-

ti cations with distinct payload strings will always be delivered as

distinct noti cations. Similarly, noti cations from di ferent trans-

actions will never get folded into one noti cation. Except for drop-

ping later instances of duplicate noti cations, NOTIFY guarantees

that noti cations from the same transaction get delivered in the or-

der they were sent. It is also guaranteed that messages from di fer-

ent 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 x 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 noti cations:

```
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 main-

taining our *twcache.counters* cache in an event driven fashion.

PostgreSQL LISTEN and NOTIFY support is perfect for maintaining a cache.

Because noti cations 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 noti cations from and is-

sue the *listen* command.

2. Fetch the current values from their *single source of truth* and reset the cache with those computed values.

3. Process noti cations as they come and update the in-memory cache, and

once in a while synchronize the in-memory cache to its materialized loca-

tion, 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 noti cations 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:

```
begin;


```

```
create schema if not exists twcache;

create table twcache.counters
(
messageid bigint not null primary key,
rts bigint,
favs bigint
);

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 t 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 e cient 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 nd 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 de ned 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 e fort” and may not result in desired values for some

types. JSON keys are matched to identical column names in the

target row type. JSON elds that do not appear in the target row

type will be omitted from the output, and target columns that do

not match any JSON eld will simply be NULL.

The function allows transforming a JSON document into a full-blown rela-

tional 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 ma-

terialize a full cache as fetched at startup, and to materialize the cache we build

in-memory while receiving noti cations.

The query used to fetch the initial value of the cache and set it again at startup is the following:

```
select messageid, rts, favs
from tweet.message_with_counters;
```

We use the view de nition that we saw earlier to do the computations for us, and

ll 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
```

5

Starting benchmark for inserts

6

Inserting took 10.351908 seconds, did 10000 rts

Remember when reading those numbers that we can’t compare them mean-

ingfully anymore. We installed our trigger af er 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, noti cations 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 noti cations with a JSON payload

and then reported the cumulated gures 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 rst 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 noti cation capabilities

are capable of missing events. Noti cations 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 Post-

greSQL™ Extensions to the JDBC API, and quoting their page: A key limitation of the JDBC driver is that it cannot receive asynchronous noti cations and must poll the backend to check if any

noti cations were issued. A timeout can be given to the poll func-

tion, 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 noti cations 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 noti cations 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.

40

Batch Update, MoMA Collection

The Museum of Modern Art (MoMA) Collection hosts a database of the mu- seum’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, rep-

resenting 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, dimen-
sions, and date acquired by the Museum. Some of these records
have incomplete information and are noted as “not Curator Ap-
proved.”

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.

```
begin;

create schema if not exists moma;

create table moma.artist
(
  constituentid integer not null primary key,
name text not null,
bio text,
nationality text,
```

```
gender text,
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

Af er 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 le with a new content.

A *batch update* operation is typically implemented that way:

• Load the new version of the data from le to a PostgreSQL table or a *tem-*

porary 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:

1

```
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 speci cations 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 rst 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: 1

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 rst update script and the second one, that handles concurrency con icts:

1

--- 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,
(select count(*) from ins) as inserts;
```

Notice the new line *on conflict (constituentid) do nothing*. It basically implements what it says: if inserting a new row causes a con ict, then the operation

for this row is skipped.

The con ict 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 ight, 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 al-

ways want correctness and clarity. That mindset dictat how I ap-

proach concurrency problems. I worry about things like transaction

boundari 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 issu rarely crop up where you predict — it's better to *observe what's really happening. Same with scaling issu — you*

might think you know which parts of the system will be in high

demand, but reality will often surprise you. But if you foc

on get-

ting 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 h two sid — " it right?" and "

it fast?" — and I worry about the first one first. As Paul Phillips

rightly said, performance tail, correctness 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 di ferent 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

to

bring some clarity to how we deal with the effect of time in program-

ming. Clojure's key insight

that time doesn't just complicate con-

current database transactions, but nearly every aspect of program-

ming. Concurrency , "what happens if someone else stomps over

my data at the same time?" Mutability , "what happens if some-

one else stomps over my data, or if I stomp on it myself?" Languag

with immutable data structur , like Clojure, ask, "Why don't we

just eliminate that whole problem?"

So Clojure

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

no concurrency, there

are no competing timelin , 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 fre you up

from worrying about concurrency and mak you very mindful of

it.

Haskell I'd say tak that even further. It doesn't just make you *suspicio*

of the side-effects time

having on your code, but just

about all side effects. Haskell’s been ferocio

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 languag beat out side effects and then gradually bring

them back in, with controls. And what controls do we see for concur-

rency? For the most part, it’s not the low-level locks and semaphor

of C and Java, but the higher-level ide we love from databas , 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 h

two key rol for me. First,

the storage engine. It’s the

golden record of what our system knows. Every important fact

should be there. That probably mak

the database the most

precio

part of the system, but that’s okay. Data

precio .

The other role

much more abstract. I use the database a design

tool. There’s something great about the relational mindset that en-

courag you to think about data in itself, separate from how it’s

used. You model the data so it tak its own real shape, rather than

the shape today’s task wants it to have.

By way of contrast, I think one of the downsid of test-driven-

development

in some corners it’s encouraged people to think of

their data

a kind of black box, where only the way it’s used to-

day gets to drive the data implementation. Too often I’ve seen that

lead to big painful rewrit when the data outgrows the featur .

The mindset of making data primary, and today’s use-case sec-

ondary, invaluable if you want a system to grow well. And that’s

something Codd figured out decad ago.

I w 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 stru led with, without

breaking a sweat. They taught me early on that if you get your data

model right, every feature easier. You can get things right without

trying while you competitors firefight their mistak .

When using PostgreSQL, do you pick the default isolation level or do you have a

speci c 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 tim , for a couple of extremely

specific cas .

Part VIII

PostgreSQL is unique in its approach to data types. The initial design of Postgres

can be read about in the document entitled The Design Of Postgres, authored by Michael Stonebraker and Lawrence A. Rowe.

Quoting this foundation paper, we can read:

Th paper presents the preliminary design of a new database man-

agement system, called POSTGRES, that the successor to the IN-

GRES relational database system. The main design goals of the

new system are to:

1. *Provide better support for complex objects,*

2. *Provide user extendibility for data typ , operators and access*

methods,

3. *Provide faciliti for active databas (i.e. alerters and tri ers)*

and inferencing including forward- and backward-chaining,

4. *Simplify the DBMS code for crash recovery,*

5. *Produce a design that can take advantage of optical disks,*

workstations composed of multiple tightly-coupled processors,

and custom designed VLSI chips, and

6. *Make*

few chang

possible (preferably none) to the rela-

tional model.

The paper describ the query language, programming language in-

terface, system architecture, query processing strate , and storage

system for the new system.

Current modern version of PostgreSQL still follow several of the same design

rules. The development team managed to improve the many facets of the system,

including adding a full implementation of the SQL standard, without having to

change the extensibility foundations of Postgres.

In this chapter, we are going to learn about some advanced extensions for Post-

greSQL, distributed as part of the *contrib* distribution or by developers other than PostgreSQL itself.

What’s a PostgreSQL Extension?

A PostgreSQL extension is a set of SQL objects that you can add to PostgreSQL

catalogs. Installing and enabling an extension can be done at run-time, making

deploying extensions as simple as typing a single SQL command.

PostgreSQL extensions are available to cover di ferent needs, such as the follow-

ing non-exhaustive list:

• Extensions for application developers

These extensions typically introduce an augmented feature set to

PostgreSQL, making new specialized tricks available to your SQL queries.

Examples of such extensions include PostGIS, a spatial database extender for the PostgreSQL object-relational database that adds support for geographic objects allowing location queries to be run in SQL.

• Extensions for PostgreSQL service administrators (ops, dba)

These extensions typically introdude new introspection facilities or useful

tooling to administer your PostgreSQL production instances.

Examples of such extensions include pageinspect, which provides functions that allow you to inspect the contents of database pages at a low level,

which is useful for debugging purposes.

• Extensions for pluggable languages

These extensions typically implement support for a programming

language to be used for writing stored procedures and functions.

PostgreSQL maintains several procedural languages in-core:

– PL/C of course

– PL/SQL, which allows it to use a SQL query with parameters: it is not really procedural because it’s plain SQL wrapped in a function

de nition

– PL/pgSQL, a procedural language that implements SQL as a rst-class citizen and provides procedural control structures around SQL

statements

– PL/TCL which allows using the TCL programming language to write stored procedures and functions

– PL/Perl

– PL/Python

Adding to that list, we can nd other programming languages support in

external projects — i.e. they’re not maintained by the PostgreSQL com-

mitters team. For instance Plv8 embeds server-side Javascript code right into your database server, then there’s PL/Java, PL/Lua and many others.

• Extensions for foreign data wrappers

These extensions typically implement support for a accessing data man-

aged externally to PostgreSQL, following the SQL/MED design, which is part of the SQL standard. In SQL/MED, MED stands for *management*

of external data.

PostgreSQL ships with some Foreign Data Wrappers that allow it to read

data from les with le_fdw or from a remote PostgreSQL server with

postgres_fdw.

Other FDWs can be found that are not maintained by the PostgreSQL committers team, such as oracle_fdw or ldap_fdw. The list is incredibly long and diverse, so be sure to check out the foreign data wrappers page on the PostgreSQL wiki.

As you can see from this rough categorization attempt, PostgreSQL extensions

can implement a very wide variety of tools and enhancements.

Inside PostgreSQL Extensions

Any SQL object can be part of an extension, and here's a short list of common

objects found in popular extensions:

- Stored procedures
- Data type
- Operator, operator class, operator family
- Index access method

As an example, we install the pg_trgm contrib extension and have a look at what it contains:

```
create extension pg_trgm;
```

Now the extension is enabled in my database, and it's possible to list the object

contained in the pg_trgm extension thanks to the psql command \dx+ pg_trgm.

Here’s the output of the command:

```
Objects in extension "pg_trgm"

Object description

════════════════════════════════════════════════════════════

function
gin_extract_query_trgm(text,internal,smallint,internal,internal,internal,interna
function gin_extract_value_trgm(text,internal)

function
gin_trgm_consistent(internal,smallint,text,integer,internal,internal,internal,inte
function
gin_trgm_triconsistent(internal,smallint,text,integer,internal,internal,internal)
function gtrgm_compress(internal)

function gtrgm_consistent(internal,text,smallint,oid,internal)

function gtrgm_decompress(internal)

function gtrgm_distance(internal,text,smallint,oid,internal)

function gtrgm_in(cstring)

function gtrgm_out(gtrgm)

function gtrgm_penalty(internal,internal,internal)

function gtrgm_picksplit(internal,internal)

function gtrgm_same(gtrgm,gtrgm,internal)

function gtrgm_union(internal,internal)

function set_limit(real)

function show_limit()
function show_trgm(text)
function similarity(text,text)
function similarity_dist(text,text)
function similarity_op(text,text)
function word_similarity(text,text)
function word_similarity_commutator_op(text,text)
function word_similarity_dist_commutator_op(text,text)
function word_similarity_dist_op(text,text)
function word_similarity_op(text,text)
operator %(text,text)
operator %>(text,text)
operator <%(text,text)
operator <->(text,text)
```

```
operator <->>(text,text)
operator <<->(text,text)
operator class gin_trgm_ops for access method gin
operator class gist_trgm_ops for access method gist
operator family gin_trgm_ops for access method gin
operator family gist_trgm_ops for access method gist
```

```
type gtrgm
(36 rows)
```

The functions listed here are stored procedure, and in this extension they hap-

pen to be written in C. Then we see several new operators such as %, which im-

plements a similarity test. We’re going to cover that in detail later in this chapter.

The *operator class* and *operator family* entries can be considered as glue objects.

They register index access methods covering the operators provided in the Post-

greSQL catalogs, so that the planner is capable of deciding to use a new index.

Finally, the extension implements a new datatype that is also implemented in C

and installed at run-time, without having to recompile the PostgreSQL server or

even restart it, in this case.

Installing and Using PostgreSQL Extensions

PostgreSQL extensions live in a given database, even when their deployment in-

cludes shared object libraries that are usually system wide. Depending on your

operating system, a shared object might be a .so le, or a .dll le, or even a

.dylib le.

Once the support les for an extension are deployed at the right place on your

operating system, we can type the following SQL command to enable the trigram

extension in the current database we are connected to:

```
create extension pg_trgm ;
```

Installing the support les for an extension is done via installing the proper pack-

age for your operating system. When using Debian make sure to check out the PostgreSQL Debian distribution at http://apt.postgresql.org.

To make pg_trm installable in PostgreSQL we have to install the proper contrib

package, which is easily done in Debian, as in the following example where we

are targeting PostgreSQL version 10:

```
$ sudo apt-get install postgresql-contrib-10
```

It is possible to check whether an extension has already been made available to

your PostgreSQL instance with the following SQL query:

```
table pg_available_extensions;
```

Here’s an example list:

```
name
│ default_version │ installed_version │
comment
════════════════════╪════════════════════╪══════════
pg_prewarm
│ 1.1
│ ¤
│ prewarm relation data
pgcrypto
│ 1.3
│ ¤
│ cryptographic functions
lo
│ 1.1
│ ¤
│ Large Object maintenance
plperl
│ 1.0
```

│ ¤

│ PL/Perl procedural language

pgstattuple

│ 1.5

│ ¤

│ show tuple-level statistics

plpgsql

│ 1.0

│ 1.0

│ PL/pgSQL procedural language

tcn

│ 1.0

│ ¤

│ Triggered change notifications

pg_buffercache

│ 1.3

│ ¤

│ examine the shared buffer cache

pg_freespacemap │ 1.2

│ ¤

│ examine the free space map (FSM)

sslinfo

│ 1.2

│ ¤

│ information about SSL certificates

(10 rows)

Finding PostgreSQL Extensions

The rst set of interesting extensions that should be available on any PostgreSQL

installation is the contribs themselves. Make sure the operating system package

for contribs is always deployed everywhere you’re using PostgreSQL, so that you

can then put those wonderful extensions to good use.

Some of the contrib extensions are meant to debug hairy situations, and you’ll

be happy that diagnostics are only a create extension command away when

you need to nd out if a table or an index is corrupted, for instance.

Another source of PostgreSQL extensions is the PostgreSQL Extension Network

where extension authors can register their project themselves, and update the in-

formation when they release new versions.

In both cases, there's no guarantee of the quality of any of the extensions listed, so you will have to test them yourself. In this book we're going to cover extensions

that have been known to be of production quality, i.e. the ones that you can rely

on. We're also going to add a list of trustworthy extensions even if we don't cover

them in details. The list is not exhaustive though, so if you nd an extension not

listed on these pages, it's most certainly worth a try!

A Primer on Authoring PostgreSQL Extensions

PostgreSQL makes it easy to author an extension. While most extensions need

to be written in C in order to have access to low-level PostgreSQL facilities, it's

not always the case and some extensions can be written in other higher order

programming languages such as PL/Perl, PL/Python or even PL/pgSQL.

If your application already maintains parts of its logic in stored procedures,

you might nd it useful to rely on the PostgreSQL extension facility. The

PostgreSQL documentation section titled Extension Building Infrastructure

details the steps to follow in order to cook your own extension.

You will need to prepare the following les:

• Make le, if you need to “build” your les, which is mostly necessary when

writing an extension in C

• Control le, to describe the extension properties

• SQL script that is played to install the extension objects, such as tables,

views, functions, stored procedures, operators, data types, etc.

• SQL upgrade scripts to go from one version to the next

If you’re already managing stored procedures, have a look at how to ship them

to PostgreSQL as extensions. Remember that there was only one reason why ex-

tensions were added to PostgreSQL in 9.1: being able to seamlessly pg_dump and

pg_restore your database when it’s using an external module. I know because I

wrote the PostgreSQL extension feature and got this patch committed.

A Short List of Noteworthy Extensions

Here’s a list of noteworthy PostgreSQL extensions for application developers.

The following extensions add new features to your RDBMS so that you can solve

more use cases right inside the database.

Having more data processing tools in the database server is a good thing when you have complex problems to solve and want to have a solution that is both

correct (from a transactional standpoint) and e cient (from a data ow stand-

point). We'll see several detailed examples of these points in the following sec-

tions of this chapter.

Here's a list of PostgreSQL contrib extensions for application developers:

• Bloom Index Filters

Bloom provides an index access method based on bloom lters.

From the PostgreSQL documentation about this contrib extension:

A Bloom filter a space-efficient data structure that used to test whether

an element

a member of a set. In the case of an index access method,

it allows fast exclusion of non-matching tupl via signatur whose size

determined at index creation.

A signature

a lossy representation of the indexed attribute(s), and

such prone to reporting false positiv ; that , it may be reported that an

element in the set, when it not. So index search results must always be

rechecked using the actual attribute valu from the heap entry. Larger

signatur reduce the odds of a false positive and th *reduce the number*

of useless heap visits, but of course also make the index larger and hence

slower to scan.

Th type of index

most useful when a table h

many attribut and

queri test arbitrary combinations of them. A traditional btree index

faster than a bloom index, but it can require many btree index to

support all possible queri where one needs only a single bloom index.

Note however that bloom index only support equality queri , where

btree index can also perform inequality and range search .

• earthdistance

The earthdistance module provid two different approach to calculating

great circle distanc on the surface of the Earth. The one described first

depends on the cube module (which must be installed before earthdistance

can be installed). The second one

based on the built-in point data type,

using longitude and latitude for the coordinat .

In th module, the Earth

assumed to be perfectly spherical. (If that's *too inaccurate for you, you might want to look at the PostGIS project.)*

• hstore

Th module implements the hstore data type for storing sets of key/value

pairs within a single PostgreSQL value. Th can be useful in vario

scenarios, such

rows with many attribut that are rarely examined, or

semi-structured data. Keys and valu are simply text strings.

• ltree

Th module implements a data type ltree for representing labels of data

stored in a hierarchical tree-like structure. Extensive faciliti for searching

through label tre are provided.

And here's an example that comes straight from the documentation too,

so that you can decide if you want to have a closer look at it:

```
ltreetest=> SELECT path FROM test WHERE path @ 'Astro* & !pictures@';
path
------------------------------------
```

Top.Science.Astronomy

5

Top.Science.Astronomy.Astrophysics

6

Top.Science.Astronomy.Cosmology

7

(3 **rows**)

• pg_trgm

Th module provid functions and operators for determining the simi-

larity of alphanumeric text based on trigram matching,

well

index

operator class that support fast searching for similar strings.

Now, the next part of the list includes extensions to PostgreSQL that are main-

tained separately from the main project. That means the projects have their own

team and organization, and more importantly, their own release cycle.

• PostGIS

PostGIS

a spatial database extender for PostgreSQL object-relational *database. It adds support for geographic objects allowing location queri*

to be run in SQL.

```sql
SELECT superhero.name
FROM city, superhero
WHERE ST_Contains(city.geom, superhero.geom)
AND city.name = 'Gotham';
```

In addition to basic location awareness, PostGIS offers many featur rarely

found in other competing spatial databas such

Oracle Locator/Spatial

and SQL Server. Refer to PostGIS Feature List for more details.

• ip4r

IPv4/v6 and IPv4/v6 range index type for PostgreSQL

While PostgreSQL already h builtin typ ‘inet’ and ‘cidr’, the authors

of th module found that they had a number of requirements that were

not addressed by the builtin type.

Firstly and most importantly, the builtin typ do not have good support for index lookups of the form (column >>= parameter), i.e. where you have a table of IP address rang and wish to find which on include a given IP address. Th requir an rtree or gist index to do efficiently, and also requir a way to represent IP address rang that do not fall precisely on CIDR boundari .

Secondly, the builtin inet/cidr are somewhat overloaded with semantics, with inet combining two distinct concepts (a netblock, and a specific IP within that netblock). Furthermore, they are variable length typ (to support ipv6) with non-trivial overheads, and the authors (whose applications mainly deal in large volum of single IPv4 address) wanted a more lightweight representation.

• citus

Cit horizontally scal PostgreSQL across commodity servers using sharding and replication. Its query engine paralleliz incoming SQL queri across these servers to enable real-time respons on large datasets.

• pgpartman

pg_partman

an extension to create and manage both time-based and

serial-based table partition sets. Native partitioning in PostgreSQL 10 *supported*

of pg_partman v3.0.1. Note that all the featur of tri er-

based partitioning are not yet supported in native, but performance in

both reads and writ

significantly better.

Child table creation all managed by the extension itself. For non-native,

tri er function maintenance

also handled. For non-native partition-

ing, tabl with existing data can have their data partitioned in easily

managed smaller batch . For native partitioning, the creation of a new

partitioned set

required and data will have to be migrated over sepa-

rately.

• postgres-hll

Th Postgr module introduc a new data type hll, which

a Hyper-

LogLog data structure. HyperLogLog a fixed-size, set-like structure used

for distinct value counting with tunable precision. For example, in 1280

byt hll can estimate the count of tens of billions of distinct valu with *only a few percent error.*

• pre x

Prefix matching both very common and important in telephony applications, where call routing and costs depend on matching caller/callee phone numbers to an operator prefix.

Let’s say the prefix table

called prefix , a typical query will try to

match a phone number to the longest prefix in the table:

```
SELECT *
FROM prefixes
WHERE prefix @> '0123456789'
ORDER BY length(prefix) DESC
LIMIT 1;
```

• madlib

Apache MADlib an open-source library for scalable in-database analyt-

ics. It provid data-parallel implementations of mathematical, statistical

and machine learning methods for structured and unstructured data.

The MADlib mission: to foster widespread development of scalable an-

alytic skills, by harnessing efforts from commercial practice, academic re-

search, and open-source development.

• RUM

The RUM module provid an access method to work with RUM index.

It

based on the GIN access methods code. RUM solv the GIN rank-

ing, phrase search, and ordering by timestamps performance problems of

GIN by storing additional information in a posting tree. Positional infor-

mation of lexem or timestamps are exampl .

If you're using full text search with PostgreSQL, then have a look at the

RUM extension.

From this list it's quite clear how powerful the PostgreSQL extensibility charac-

teristics are. We have extensions that provide a new data type and its operators,

moreover with indexing support. Other extensions implement their own SQL

planner and optimizer, like in the case of Citus, which uses that capability to then route query executions over a network of distributed PostgreSQL instances.

All those PostgreSQL extension can rely on PostgreSQL industry strenghs:

• Correctness via transaction semantics

• Durability and crash safety

• Performance thanks to an advanced planner and cost-based optimizer

• Open source project and protocol

43

Auditing Changes with hstore

The PostgreSQL extension hstore implements a data type for storing sets of key/value pairs within a single PostgreSQL value. This can be useful in

various scenarios, such as rows with many attributes that are rarely examined, or

semi-structured data. Keys and values are simply text strings.

We could go so far as to say that hstore is a precursor to JSON support in Post-

greSQL, as it supports some of the same use cases. The main di ference between

hstore and JSON is that in hstore, there’s only one data type supported and

that’s text. Also, an hstore composite value is a at dictionnary, so nesting isn’t

supported.

Still hstore is very useful in some cases, and we’re going to see how to put hstore

into practice to audit changes in a generic way.

Introduction to *hstore*

Of course the rst thing we have to do is to enable the hstore extension in our

database with the following SQL command:

```
create extension hstore;
```

Now, equiped with the extension, we can create hstore values and use the arrow

operator -> to access the values associated with a given key.

```
select kv,
kv->'a' as "kv -> a",
kv-> array['a', 'c'] as "kv -> [a, c]"
from (
```

values ('a=>1,a=>2'::hstore),

6

('a=>5,c=>10')

7

)

8

as t(kv);

Here, we fetch the value from the key 'a' as a scalar value, and then we fetch the

values from multiple keys at once, with the notation array ['a', 'c']:

```
kv

│ kv -> a │ kv -> [a, c]

══════════════════════════════╪══════════════╪═════════════════

"a"=>"1"

│ 1

│ {1,NULL}

"a"=>"5", "c"=>"10" │ 5

│ {5,10}

(2 rows)
```

As you can see, all we have in hstore keys and values are *text* values.

Comparing hstores

The *hstore* extension implements a - operator: its documentation says that it will *delete matching pairs from left operand*.

```
select
'f1 => a, f2 => x'::hstore
- 'f1 => b, f2 => x'::hstore
as diff;
```

This gives the following result:

```
diff
═══════════════
"f1"=>"a"
(1 row)
```

That’s what we’re going to use in our *chang auditing tri er* now, because it’s a pretty useful format to understand what did change.

Auditing Changes with a Trigger

First we need some setup:

• We are going to track changes made when we update the MoMA collection, which we processed in the previous chapter. The table we are auditing is moma.artist.

• The changes are recorded in a table named moma.audit, de ned in a pretty generic way as we can see below.

• Then we install PostgreSQL triggers on the moma.artist table to capture any change made to it and populate the moma.audit table with the *before* and *after* versions of updated rows.

The representation of the row is recorded using the hstore format, which is very exible and could be used to track more than one table de nition. Either several tables, or just the same table even in the case of schema changes done with ALTER TABLE.

The idea is to add a row in the audit table each time the moma.artist table is updated, with the hstore representation of the data in ight before and af er the change:

```
begin;

create table moma.audit
(
change_date timestamptz default now(),
before
hstore,
after
hstore
);


commit;
```

In the previous chapter we had an introduction to triggers. Here’s an hstore auditing one:

```
begin;


create function moma.audit()
```

```
returns trigger
language plpgsql
as $$
begin
INSERT INTO audit(before, after)
SELECT hstore(old), hstore(new);
return new;
end;
$$;

create trigger audit
```

```
after update on moma.artist
for each row
execute procedure audit();

commit;
```

Note that we could attach the same trigger to any other table, as the details of the audit table contain nothing speci c about the moma.artist table. When doing so, it then becomes necessary to also track the origin of the changes with both a table_name column and a schema_name column:

```
begin;


create table moma.audit
```

```
(
change_date timestamptz default now(),
schema_name name,
table_name name,
before hstore,
after hstore
);

commit;
```

Within the trigger procedude, the information we want is available as the

TG_TABLE_SCHEMA and TG_TABLE_NAME variables. To enhance the trigger proce-

dure code that we’re using in this examples, read the PostgreSQL documentation

chapter entitled PL/pgSQL Trigger Procedures.

Testing the Audit Trigger

With that in place, let’s try it out:

```
begin;

create temp table batch
(
like moma.artist
including all
)
```

```
on commit drop;

\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)
= (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
)
on conflict (constituentid) do nothing
returning artist.constituentid
)
select (select count(*) from upd) as updates,
(select count(*) from ins) as inserts;
```

```
commit;
```

This SQL statement outputs the following information:

```
BEGIN
CREATE TABLE
COPY 15226
updates │ inserts
════════╪═════════
52 │
61
(1 row)

COMMIT
```

And thanks to our audit trigger, we can have a look at what has changed:

```
select (before -> 'constituentid')::integer as id,
after - before as diff
from moma.audit
```

```
4
limit 15;
```

So here are the rst 15 changes out of the 52 updates we made:

```
id
│
diff
═════════╪══════════════════════════════════════════════════
546 │ "bio"=>"American, born England. 1906–1994"
570 │ "bio"=>"American, 1946–2016"
920 │ "bio"=>"American, born Switzerland. 1907–1988", "end"=>"1988"
957 │ "bio"=>"Italian, 1906–1996", "end"=>"1996"
1260 │ "bio"=>"American, 1923–2017", "end"=>"2017", "begin"=>"1923"
1372 │ "bio"=>"Belgian, 1901–1986", "end"=>"1986", "name"=>"Suzanne va…
│…n Damme", "begin"=>"1901", "nationality"=>"Belgian"
1540 │ "bio"=>"American, 1900–1979", "end"=>"1979", "begin"=>"1900", "…
│…nationality"=>"American"
1669 │ "name"=>"Dušan Džamonja"
1754 │ "name"=>"Erró (Gudmundur Gudmundsson)"
1855 │ "bio"=>"Mexican, 1904–1972", "end"=>"1972"
```

```
1975 │ "bio"=>"American, born Uruguay. 1919–2013"
2134 │ "bio"=>"Israeli, 1936–2017"
2679 │ "bio"=>"British, 1932–2017"
3005 │ "bio"=>"French, 1906–1971"
3230 │ "bio"=>"Greek, 1936–2017"
(15 rows)
```

From hstore Back to a Regular Record

The hstore extension is able to cast data from a record to an hstore with the hstore() function, and back again with the populate_record() function.

Here’s an example using that very powerful function, where we nd out if any artist name has been changed and display when the change occurred, what the old name was and what the new name is:

```
select audit.change_date::date,
artist.name as "current name",
before.name as "previous name"


```

```
from
moma.artist
join moma.audit
on (audit.before->'constituentid')::integer
= artist.constituentid,
populate_record(NULL::moma.artist, before) as before
```

```
where artist.name <> before.name;
```

In this query, we extract the constituentid from the audit table in order to join it with artist table, and then build the following result set:

```
change_date │
current name
│
previous name
```

2018-08-25

│ Suzanne van Damme

│ Elisabeth van Damme

2018-08-25

│ Dušan Džamonja

│ Dusan Dzamonja

2018-08-25

│ Erró (Gudmundur Gudmundsson) │ Erro (Gudmundur Gudmundsson)

2018-08-25

│ Nikos Hadjikyriakos-Ghika

│ Nikos HadjiKyriakos-Ghika

2018-08-25

│ Sam Mendes

│ Same Mendes

2018-08-25

│ Tim Berresheim

│ Tim Berrescheim

2018-08-25

│ Kestutis Nakas

```
 │ Kęstutis Nakas
2018-08-25
 │ Jennifer T. Ley
 │ Jennifer Ley
(8 rows)
```

The hstore extension is very useful, even with JSON support in current versions

of PostgreSQL. The ability to cast from and to a record is unique to this exten-

sion, and its di ference operator has no equivalent in the JSON feature set.

Last.fm Million Song Dataset

In the next two study cases, we're going to play with the LastFm dataset, the o cial song tag and song similarity dataset of the Million Song Dataset:

The MSD team is proud to partner with Last.fm in order to bring

you the largest research collection of song-level tags and precom-

puted song-level similarity. All the data is associated with MSD

tracks, which makes it easy to link it to other MSD resources: au-

dio features, artist data, lyrics, etc.

First, we need to import this dataset into a PostgreSQL database. The data set

is o fered both as an SQLite database and a JSON le. Loading the SQLite database is easy thanks to pgloader:

```
$ curl -L -o /tmp/lastfm_tags.db
http://labrosa.ee.columbia.edu/ \
millionsong/sites/default/files/lastfm/lastfm_tags.db


$ pgloader /tmp/lastfm_tags.db pgsql://appdev@localhost/appdev
```

We get the following output, meaning the data is now available in our

PostgreSQL database for further indexing:

```
table name
errors
read
imported
bytes
total time
-----------------------
---------
```

```
---------

---------

---------

--------------

fetch

0

0

0

0.000s

fetch meta data

0

8

8

0.028s

Create Schemas

0

0

0

0.000s

Create SQL Types
```

```
0

0

0

0.006s

Create tables

0

6

6

0.031s

Set Table OIDs

0

3

3

0.009s

----------------------

---------

---------

---------

---------

-------------
```

tids

0

505216

505216

9.2 MB

1.893s

tags

0

522366

522366

8.6 MB

1.781s

tid_tag

0

8598630

8598630

135.7 MB

32.614s

```
---------

---------

---------

---------

--------------

COPY Threads Completion

0

4

4

34.366s

Create Indexes

0

5

5

2m14.346s

Index Build Completion

0

5

5

36.976s
```

Reset Sequences

0

0

0

0.054s

Primary Keys

0

0

0

0.000s

Create Foreign Keys

0

0

0

0.000s

Create Triggers

0

0

0

0.001s

```
Install Comments
0
0
0
0.000s
-----------------------
---------
---------
---------
---------
--------------
Total import time
✓
9626212
9626212
153.4 MB
3m25.743s
```

Here, *pgloader* extracted the table and index de nitions from the SQLite database using the sqlite_master catalog and the PRAGMA table_info() commands, and it migrated the data in a streaming fashion to PostgreSQL, using the *COPY protocol*.

Having a look at the *demo_tags.py* script from the Last.fm project, we can see how to use the relations here, and we realize they are using the 64-bit signed inte-

ger ROWID system column. We need something comparable to be able to make sense of the data:

```
begin;


alter table tags add column rowid serial;
alter table tids add column rowid serial;


commit;
```

With the new columns in place, we can have a rst look at the provided data. To

get started, we can search for *Brian Setzer* in the user-de ned tags:

```
select tags.tag, count(tid_tag.tid)

```

```
from tid_tag, tags
where tid_tag.tag=tags.rowid and tags.tag ~* 'setzer'
group by tags.tag;
```

Sure enough, some fans have been using Last.fm services:

```
tag
│ count
════════════════════════════════════════╪══════════
Brian Setzer
│
1
Setzer
│
13
brain setzer orchestra
│
2
brian setzer is GOD
│
```

1

brian setzer orchestra

│

3

rockabilly Setzer style

│

4

setzer is a true guitarhero │

9

the brian setzer orchestra

│

1

(8 rows)

Time: 394.927 ms

Here the query is mainly doing a join in between the *tid* table (containing track ids) and the *tid_tag* table (containing the association between tracks and tags), ltering on the *case insensitive regular expression* 'setzer'. As we can imagine from reading the query execution time, there’s no index to implement the ltering here.

Now the million song project is also releasing the data as a set of JSON-encoded text les, and in the JSON le we nd additional information such as titles and

artist that we could add to the current track table containing only the track id

information. A track id looks like TRVBGMW12903CBB920 — this is not
the

best way to refer a song for us human beings.

So this time we download the JSON resource and process it with the help of a

small parser script:

```
curl -L -o /tmp/lastfm_subset.zip
http://labrosa.ee.columbia.edu/ \
millionsong/sites/default/files/lastfm/lastfm_subset.zip
```

Then we can load this new content into the new table de nition:

```
begin;


create table lastfm.track

```

```
(
tid text,
artist text,
title text
);

commit;
```

Because my favorite programming environment involves Common Lisp, the following source of the script is written in this language. I’ve been using it to parse the JSON les from the zip archive and load them all from a COPY command.

Using COPY here means that we can stream the parsed data as we go, and inject all the content in a single PostgreSQL command:

```
(defpackage #:lastfm
(:use #:cl #:zip)
(:import-from #:cl-postgres
#:open-db-writer
#:close-db-writer
```

```
#:db-write-row))

(in-package #:lastfm)

(defvar *db* '("appdev" "appdev" nil "localhost" :port 5432))
```

```
(defvar *tablename* "lastfm.track")
(defvar *colnames*
'("tid" "artist" "title"))

(defun process-zipfile (filename)
"Process a zipfile by sending its content down to a PostgreSQL table."

(pomo:with-connection *db*

(let ((count 0)
(copier (open-db-writer pomo:*database* *tablename* *colnames*)))

(unwind-protect
(with-zipfile (zip filename)
(do-zipfile-entries (name entry zip)
(let ((pathname (uiop:parse-native-namestring name)))
(when (string= (pathname-type pathname) "json")
(let* ((bytes
(zipfile-entry-contents entry))
(content
(babel:octets-to-string bytes :encoding :utf-8)))
(db-write-row copier (parse-json-entry content))
(incf count))))))
```

```
(close-db-writer copier))

;; Return how many rows we did COPY in PostgreSQL
count)))

(defun parse-json-entry (json-data)
(let ((json (yason:parse json-data :object-as :alist)))
(list (cdr (assoc "track_id" json :test #'string=))
(cdr (assoc "artist" json :test #'string=))
(cdr (assoc "title" json :test #'string=)))))
```

Of course it’s possible to implement the same technique in any programming language. All you need is for your PostgreSQL driver of choice to expose the

PostgreSQL COPY protocol. Make sure it does, and then learn how to properly

load data using it.

With the Postmodern driver for Common Lisp that I'm using, the COPY API involves the three functions below:

• open-db-writer to open the COPY streaming protocol,

• db-write-row to push a single row to PostgreSQL,

• close-db-writer to signal we're done and close the COPY streaming.

So if you read the script carefully you'll see that it is using those API calls to

push one row per JSON le that is parsed. One trick the script is using is that it's

reading directly from the zip le, uncompressing it in memory and parsing JSON

les from there, without writing the JSON les extracted from the zip archive on

disk on the *client side*. PostgreSQL of course will have to serialize the data to disk when it appears in the *server side* of the COPY protocol.

Time to discover the data model and the data itself with a rst batch of interactive

queries, with the sole aim of ful lling our curiosity:

```
select artist, count(*)
```

```
from lastfm.track
group by artist
order by count desc
limit 10;
```

We can see that one of the most popular artists in the data set is Aerosmith:

```
artist
│ count
═══════════════════════════════════════╪══════════
Mario Rosenstock
│
13
Aerosmith
│
12
Snow Patrol
│
```

12

Phil Collins

|

12

Sugar Minott

|

11

Bill & Gloria Gaither

|

11

Line Renaud

|

11

Shakira

|

11

Radiohead

|

11

Nick Cave and the Bad Seeds |

```
(10 rows)
```

Now, let’s have a look at the kind of tags this artist would have had attached to
by Last.fm users:

```
select track.artist, tags.tag, count(*)
from tags
join tid_tag tt on tags.rowid = tt.tag
join tids on tids.rowid = tt.tid
join lastfm.track on track.tid = tids.tid
where track.artist = 'Aerosmith'
group by artist, tags.tag
order by count desc
```

```
9
limit 10;
```

With this very simple and classic query, we can see how the data model ts to-

gether, using the tags, tid_tag, tids, and track tables. The model comes from

the SQLite database used by the project, to which we have been adding the track

table, where we did COPY data from the zip le full of JSON les.

Anyway, here are some tags for Aerosmith:

```
artist
│
tag
│ count
═══════════════╪══════════════════════╪══════════
Aerosmith │ Radio4You
│
12
Aerosmith │ hard rock
│
12
Aerosmith │ rock
```

|

11

Aerosmith | classic rock

|

11

Aerosmith | 70s

|

10

Aerosmith | 80s

|

9

Aerosmith | mi metal1

|

8

Aerosmith | favorites

|

8

Aerosmith | male vocalists |

8

Aerosmith | pop

```
|
8
(10 rows)
```

We limited it to ten rows here. The dataset we are playing with actually contains

464 unique tags just for the Aerosmith band. One of them from the list above is

spelled favorites, so what titles have been agged as a *favorite* of Last.fm users, using one spelling or another?

```
select track.tid, track.title, tags.tag
from tags
join tid_tag tt on tags.rowid = tt.tag
join tids on tids.rowid = tt.tid
join lastfm.track on track.tid = tids.tid
where track.artist = 'Aerosmith'
```

```
and tags.tag ~* 'favourite'
order by tid, tag;
```

We can see the 12 all-time favorite songs from Aerosmith… in this dataset at least:

```
tid
│
title
│
tag
═══════════════════╪═══════════════════════════
TRAQPKV128E078EE32 │ Livin' On The Edge
│ Favourites
TRAVUAJ128E078EDA2 │ What It Takes
│ favourite
TRAYKOC128F930D2B8 │ Cryin'
│ Favourites
TRAYKOC128F930D2B8 │ Cryin'
│ favourite
TRAZDPO128E078ECE6 │ Crazy
```

```
│ Favourites

TRAZDPO128E078ECE6 │ Crazy

│ all- time favourite

TRAZDPO128E078ECE6 │ Crazy

│ favourite

TRAZISI128E078EE2F │ Same Old Song and Dance │ first favourite
metalcore song

TRBARHH128E078EDE9 │ Janie's Got A Gun

│ favourite

TRBARHH128E078EDE9 │ Janie's Got A Gun

│ my favourite songs

TRBGPJP128E078ED20 │ Crazy

│ Favourites

TRBGPJP128E078ED20 │ Crazy

│ favourite

(12 rows)
```

Now that we have an idea about the dataset, it’s time to solve more interesting

use cases with it.

Using Trigrams For Typos

Some popular search engines are capable of adding helpful bits of information

that depend directly on your search phrase. Both *autocorrect* and *did you mean?*

are part of the basics of a search engine user experience nowadays.

PostgreSQL implements several *fuzzy* string matching approaches, and one of

them in particular is suitable for implementing suggestions to search strings, pro-

vided that you are searching in a known catalog of items.

The pg_trgm PostgreSQL Extension

The PostgreSQL extension pg_trgm provides functions and operators for determining the similarity of alphanumeric text based on trigram matching, as well as

index operator classes that support fast searching for similar strings.

Before we see how to bene t from the *pg_trgm* extension, it must be said that PostgreSQL comes with a complete full text search implementation. For full

exibility and advanced processing, consider using text search parsers and one of the PostgreSQL dictionnaries with support for *stemming*, *thesaur* or *syno-myms* support. The facility comes with a full text query language and tools for

ranking search result. So if what you need really is full text search then go check the docs.

The use of *trigrams* is of en complementary to *full text search*. With trigrams we can implement typing correction suggestions or index like and POSIX Regular

Expressions searches.

Whatever the use case, it all begins as usual by enabling the extension within your

database server. If you’re running from PostgreSQL packages be sure to always install the contrib package — it really is important. A time will come when you

need it and you will then be happy to only have to type create extension to

get started.

```
1

create extension pg_trgm;
```

Trigrams, Similarity and Searches

The idea behind trigrams is simple and very e fective. Split your text into a con-

secutive series of three-letters. That’s it. Then you can compare two texts based

on how many consecutives three-letters series (trigrams) are common, and that’s

the notion of similarity. It works surprisingly well, and doesn’t depend on the

language used.

In the following query we show trigrams extracted from several attempts

at spelling the name *Tommy* and then the similarity value obtained when comparing tomy and dim to tom.

```
select show_trgm('tomy') as tomy,
show_trgm('Tomy') as "Tomy",
show_trgm('tom torn') as "tom torn",
similarity('tomy', 'tom'),
similarity('dim', 'tom');
```

Note that when using small units of text the similarity might look more like a guess than anything. Also before we read the result of the query, here’s what the pg_trgm documentation says about the similarity function:

Returns a number that indicates how similar the two arguments are. The range of the result is zero (indicating that the two strings are completely dissimilar) to one (indicating that the two strings are identical).

```
-[ RECORD 1 ]------------------------------------
tomy
```

| {"

t"," to","my ",omy,tom}

Tomy

| {"

t"," to","my ",omy,tom}

tom torn

| {"

t"," to","om ",orn,"rn ",tom,tor}

similarity | 0.5

similarity | 0

As you can read in the PostgreSQL trigram extension documentation, the default similarity threshold is 0.3 and you can tweak it by using the GUC setting

pg_trgm.similarity_threshold.

Now we can search for songs about love in our collection of music, thanks to the

following query:

\index{Operators!%}

1

select artist, title

```
from lastfm.track
where title % 'love'
group by artist, title
order by title <-> 'love'
limit 10;
```

This query introduces several new operators from the pg_trgm extension:

• The operator % reads *similar to* and involves comparing trigrams of both its lef and right arguments

• The operator <-> computes the “distance” between the arguments, i.e. one minus the similarity() value.

Here’s a list of ten songs with a title similar to love:

```
artist | title
═══════╪═══════
```

```
The Opals
│ Love
YZ
│ Love
Jars Of Clay
│ Love Me
Angelo Badalamenti
│ Love Me
Barry Goldberg
│ Lost Love
The Irish Tenors
│ My Love
Jeanne Pruett
│ Love Me
Spade Cooley
│ Lover
Sugar Minott
│ Try Love
David Rose & His Orchestra │ One Love
(10 rows)
```

This *trigram similarity* concept is quite di ferent to a regexp match:

```
select artist, title
from lastfm.track
where title ~ 'peace';
```

The query above returns no rows at all, because *peace* is never found written exactly that way in the song titles. What about searching in a case insensitive way

then?

```
select artist, title
from lastfm.track
where title ~* 'peace';
```

Then we nd the following 11 titles, all embedding a variation of lower case and

upper case letters in the same order as in the expression peace:

artist	title
Bow Wow Wow	Love, Peace and Harmony
Billy Higgins	Peace
John Mellencamp	Peaceful World
Terry Riley	Peace Dance
Steinski	Silent Partner (Peace Out)
Nestor Torres	Peace With Myself
Dino	Wonderful Peace
Uman	The Way To Peace

```
Dhamika

│ Peace Prayer

Gonzalo Rubalcaba │ Peace and Quiet Time

Twila Paris

│ Perfect Peace

(11 rows)
```

Now that we have had a look at what a regexp query nds for us, we can compare

it with a trigram search.

\index{Operators!%}

```
select artist, title
from lastfm.track
where title % 'peace';
```

This query when using the ~ operator didn’t nd any titles, because peace is

always spelled with a capital letter in our catalogue. When using trigrams though,

the outcome is not so similar:

```
artist
```

```
│

title

══════════════════════╪════════════════════════

Billy Higgins

│ Peace

John Mellencamp │ Peaceful World

Terry Riley

│ Peace Dance

Nestor Torres

│ Peace With Myself

Dino

│ Wonderful Peace

Uman

│ The Way To Peace

Dhamika

│ Peace Prayer

Twila Paris

│ Perfect Peace

(8 rows)
```

Indeed, trigrams are computed in a case insensitive way:

```
select show_trgm('peace') as "peace",
show_trgm('Peace') as "Peace";
```

```
─[ RECORD 1 ]──────────────────────────────────
peace │ {"
p"," pe",ace,"ce ",eac,pea}
Peace │ {"
p"," pe",ace,"ce ",eac,pea}
```

There’s yet another way to search for similarity, called word_similarity. As per
the documentation:

This function returns a value that can be approximately understood
as the greatest similarity between the rst string and any substring
of the second string. However, this function does not add padding
to the boundaries of the extent. Thus, the number of additional
characters present in the second string is not considered, except for
the mismatched word boundry.

In other words, this function is better at nding words in a longer text. It sounds like it’s well adapted to searching our title strings, so we can try it now:

```
select artist, title
from lastfm.track
where title %> 'peace';
```

We now use a new operator: %>. This operator uses the word_similarity func-
tion introduced above, and takes into account that its lef operand is a longer
string, and its right operand is a single word to search. We could use the <% op-
erator, where lef and right operands are used the other way round: *word <%
phrase*.

Here’s what we nd this time:

```
artist
│
title
═══════════════╪═══════════════════
Bow Wow Wow
│ Love, Peace and Harmony
```

Billy Higgins

│ Peace

John Mellencamp

│ Peaceful World

Terry Riley

│ Peace Dance

Steinski

│ Silent Partner (Peace Out)

Nestor Torres

│ Peace With Myself

Dino

│ Wonderful Peace

Uman

│ The Way To Peace

Dhamika

│ Peace Prayer

Gonzalo Rubalcaba │ Peace and Quiet Time

Twila Paris

│ Perfect Peace

Dub Pistols

```
│ Peaches - Fear of Theydon remix
(12 rows)
```

Is there any di ference in what we found? Let’s write a query to nd out:

```
select artist, title
from lastfm.track
where title %> 'peace'

except

select artist, title
from lastfm.track
where title ~* 'peace';
```

PostgreSQL computes the di ference between the two result sets for us, reporting

this line:

```
artist

│

title
═══════════════════╪══════════════════════════════════════
Dub Pistols │ Peaches - Fear of Theydon remix

(1 row)
```

It seams like Peaches is similar enough to peace to be selected here.

Complete and Suggest Song Titles

Now, what if the search string is being mistyped? We all make typos, and our

users will too. Let’s try it with a small typo: peas.

```
select artist, title
from lastfm.track
where title ~* 'peas';
```

This query returns no rows! It seems our Last.fm selection of titles doesn’t include the famous *Pass The Pe* by *Maceo Parker*. Anyway, our users will not be very happy with no result, and I’m sure they would like to see suggestions of

results.

So instead we could use the following similarity query:

```
select artist, title
from lastfm.track
where title %> 'peas';
```

And now here’s a list of song titles having trigrams that are similar to the trigrams of our search string:

```
artist
│
title
══════════════════════════════╪══════════════
Bow Wow Wow
│ Love, Peace and Harmony
```

```
The Balustrade Ensemble
```

│ Crushed Pears

Little Joe & The Thrillers │ Peanuts

Billy Higgins

│ Peace

John Mellencamp

│ Peaceful World

Terry Riley

│ Peace Dance

Joe Heaney

│ Peigin is Peadar

Steinski

│ Silent Partner (Peace Out)

Nestor Torres

│ Peace With Myself

Dino

│ Wonderful Peace

Uman

│ The Way To Peace

Fania All-Stars

│ Peanuts (The Peanut Vendor)

```
Dhamika
│ Peace Prayer
Tin Hat Trio
│ Rubies, Pearls and Emeralds
Gonzalo Rubalcaba
│ Peace and Quiet Time
Twila Paris
│ Perfect Peace
Dub Pistols
│ Peaches - Fear of Theydon remix
(17 rows)
```

That’s 17 rows, so maybe too many for a suggestion as you type input box. We would like to limit it to the top ve elements, ordered by how close the titles are to the search term:

```
select artist, title
from lastfm.track
```

where title %> 'peas'

order by title <-> 'peas'

limit 5;

Here we use the <-> distance operator again in order to get this short selection: artist

│

title

═══════════════════════════╪═══════════════

Billy Higgins

│ Peace

Little Joe & The Thrillers │ Peanuts

Terry Riley

│ Peace Dance

Dhamika

│ Peace Prayer

Twila Paris

│ Perfect Peace

(5 rows)

Trigram Indexing

Of course if we want to be able to use those suggestions directly from our nice

user input facility, it needs to be as fast as possible. The usual answer to speed up speci c SQL queries is indexing.

The pg_trgm extension comes with speci c indexing algorithms to take care of

searching for similarity. Moreover, it covers searching for regular expressions too.

Here’s how to build our index:

```
create index on lastfm.track using gist(title gist_trgm_ops);
```

We can explain our previous queries and see that they now use our new index:

```
explain (analyze, costs off)
select artist, title
from lastfm.track
where title ~* 'peace';
```

Here’s the query plan:

```
QUERY PLAN
══════════════════════════════════════════════════════════════════
Index Scan using track_title_idx on track (actual time=0.552..3.832 rows=11
loops=1)

Index Cond: (title ~* 'peace'::text)

Planning time: 0.293 ms

Execution time: 3.868 ms

(4 rows)
```

What about this more complex query ordering by distance?

```
explain (analyze, costs off)
select artist, title
from lastfm.track
where title %> 'peas'
order by title <-> 'peas'
```

limit 5;

As you can see below, PostgreSQL is still able to implement it with a single index

scan. Of course the limit part of the query is done with its own query plan on

top of the index. This plan step is able to stop the index scan as soon as it has sent the rst ve rows, because the index scan is known to return them in order:

```
QUERY PLAN

Limit (actual time=6.730..6.773 rows=5 loops=1)
```

```
->

Index Scan using track_title_idx on track (actual time=6.728..6.770 rows=5
loops=1)

Index Cond: (title %> 'peas'::text)

Order By: (title <-> 'peas'::text)

Planning time: 0.090 ms

Execution time: 6.809 ms

(6 rows)
```

Finally, we can see that the query execution times obtained on my laptop are

encouraging, and we are going to be able to use those queries to serve users live.

Denormalizing Tags with intarray

Handling user-de ned tags can be challenging in SQL when it comes to allow-

ing advanced user queries. To illustrate the point here, we're going to index and

search for Last.fm tracks that are tagged as *blu* and *rhythm and blu* .

Using teh Last.fm dataset from the Million Song Dataset project provides a data set that we can reuse that is full of tracks and their user tags.

Advanced Tag Indexing

PostgreSQL comes with plenty of interesting datatypes, and one of them is known as the arrays type. PostgreSQL also provides a very rich set of extensions, some of them found under the *contrib* package; one of them is intarray. Let me quote the most interesting part of the documentation for that extension:

The @@ and ~~ operators test whether an array satis es a query,

which is expressed as a value of a specialized data type query_int.

A query consists of integer values that are checked against the

elements of the array, possibly combined using the operators &

(AND), | (OR), and ! (NOT). Parentheses can be used as needed.

For example, the query 1&(2|3) matches arrays that contain 1 and

also contain either 2 or 3.

```
create extension intarray;
```

The way the *intarray* extension works, we need to build a new table that contains the list of tags it’s been associated with for each track as an array of integers. We’re going to use our *rowid* identi er for that purpose, as in the following query:

```
select tt.tid, array_agg(tags.rowid) as tags
from
tags
```

```
join tid_tag tt
on tags.rowid = tt.tag
group by tt.tid
limit 3;
```

And here are our rst three songs with tags as numbers rather than strings:

```
tid │
tags
═══════╪═══════════════
1 │ {1,2}
2 │ {3,4}
3 │ {5,6,7,8}
(3 rows)
```

We might not want to do this computation of tags text to an array of numbers

for every title we have, so we can cache the result in a materialized view instead:

```
begin;
```

```
create view lastfm.v_track_tags as
select tt.tid, array_agg(tags.rowid) as tags
from tags join tid_tag tt on tags.rowid = tt.tag
group by tt.tid;


create materialized view lastfm.track_tags as
select tid, tags
from v_track_tags;


create index on track_tags using gin(tags gin__int_ops);


```

```
commit;
```

Given this materialized view, we are going to be able to do advanced indexing and

searching of the user provided tags. As you can see in the previous SQL script, we

have been indexing our materialized view with a special index operator, allowing

us to bene t from the intarray advanced querying.

Searches

Now we are ready for the real magic. Let’s nd all the tracks we have that have

been tagged as both *blu* and *rhythm and blu* :

```
select array_agg(rowid)
from tags
where tag = 'blues' or tag = 'rhythm and blues';
```

That query gives the following result, which might not seem very interesting at

rst:

```
array_agg

{3,739}
(1 row)
```

The intarray PostgreSQL extension implements a special kind of query string, named query_int. It looks like '(1880&179879)' and it supports the three logic operators *not*, *and*, and *or*, that you can combine in your queries.

As we want our tag search queries to be dynamically provided by our users, we are going to build the query_int string from the *tags* table itself:

```
select format('(%s)',
string_agg(rowid::text, '&')
)::query_int as query
from tags
where tag = 'blues' or tag = 'rhythm and blues';
```

This query uses the format PostgreSQL function to build a string for us, here puting our intermediate result inside parentheses. The intermediate result is ob-

tained with string_agg which aggregates text values together, using a separator

in between them. Usually the separator would be a comma or a semicolon. Here

we are preparing a query_int string, and we're going to search for all the tracks

that have been tagged both *blu* and *rhythm and blu* , so we're using the *and* operator, written &:

```
query
═════════════

3 & 739

(1 row)
```

That query here allows us to easily inject as many tags as we want to, so that it's

easy to use it as a *template* from within an application where the user is going to provide the tags list. The *intarray* extension's *query* format also accepts other

operators (*or* and *not*) as we saw before, so if you want to o fer those to your users you would need to tweak the query_int building part of the SQL.

Now, how many tracks have been tagged with *both* the *blu* and the *rhythm and blu* tags, you might be asking:

1

```
with t(query) as (
select format('(%s)',
array_to_string(array_agg(rowid), '&')
)::query_int as query
from tags
where tag = 'blues' or tag = 'rhythm and blues'
)
select count(*)
from track_tags join t on tags @@ query;
```

As you can see we use the query *template* from above in a common table expression and then inject it in the nal SQL query as join restriction over the *track_tags* table.

```
count
```

```
2278
(1 row)
```

We have 2278 tracks tagged with both the *blu* and *rhythm and blu* tags.

Now of course you might want to fetch some track meta-data, but here the only one we have is the track *hash id*:

```
with t(query) as (
select format('(%s)',
array_to_string(array_agg(rowid), '&')
)::query_int as query
from tags
where tag = 'blues' or tag = 'rhythm and blues'
)
```

```
select track.tid,
left(track.artist, 26)
|| case when length(track.artist) > 26 then '…' else " end
as artist,
left(track.title, 26)
|| case when length(track.title) > 26 then '…' else " end
as title
from
track_tags tt
join tids on tt.tid = tids.rowid
join t on tt.tags @@ t.query
join lastfm.track on tids.tid = track.tid
19
order by artist;
```

That gives us the following result:

```
tid
│
artist
│
title
══════════════════════════════╪══════════════════════════════
TRANZKG128F429068A │ Albert King
│ Watermelon Man
TRASBVS12903CF4537 │ Alicia Keys
│ If I Ain't Got You
TRAXPEN128F933F4DC │ B.B. King
│ Please Love Me
TRBFNLX128F4249752 │ B.B. King
│ Please Love Me
TRAUHJH128F92CA20E │ Big Joe Turner
```

│ Nobody In Mind

TRAOAVZ128F9306038 │ Big Joe Turner

│ Chains Of Love

TRAPRRP12903CD97E9 │ Big Mama Thornton

│ Hound Dog

TRBBMLR128F1466822 │ Captain Beefheart & His Ma… │ On Tomorrow

TRACTQD128F14B0F9D │ Donny Hathaway

│ I Love You More Than You'l…

TRAXULE128F9320132 │ Fontella Bass

│ Rescue Me

TRBAFBU128F427EFCE │ Free

│ Woman

TRAOMMU128F933878B │ Guitar Slim

│ The Things That I Used To …

TRAGVWF128F4230C95 │ Irma Thomas

│ The Same Love That Made Me…

TRACRBQ128F4263964 │ J.J. Cale

│ Midnight In Memphis

TRALIVO128F4279262 │ Janis Joplin

│ Down On Me

TRAPKJT128F9311D9E │ John Mayall & The Bluesbre… │ I'm Your Witchdoctor

TRADJGU128F42A6C00 │ Jr. Walker & The All Stars

│ Shake And Fingerpop

TRARSZV12903CDB2DE │ Junior Kimbrough

│ Meet Me In The City

TRAZANO128F429A795 │ Little Milton

│ Little Bluebird

TRBIGUJ128F92D674F │ Little Willie John

│ Leave My Kitten Alone

TRAVEOQ128F931C8F4 │ Percy Mayfield

│ Please Send Me Someone To …

TRBCGHP128F933878A │ Professor Longhair

│ Bald Head

TRALWNE12903C95228 │ Ray Charles

│ Heartbreaker

TRAGIJM12903D11E62 │ Roy Brown

│ Love Don't Love Nobody

TRBBRTY128F4260973 │ Screamin' Jay Hawkins

│ Talk About Me

TRAHSYA128F428143A │ Screamin' Jay Hawkins

```
│ I Put A Spell On You
TRAYTDZ128F93146E3 │ Stevie Ray Vaughan And Dou… │ Mary Had A Little Lamb
TRAIJLI128F92FC94A │ Stevie Ray Vaughan And Dou… │ Mary Had A Little Lamb
TRACHTO12903CBE58B │ The Animals
│ The Story of Bo Diddley
TRBFMTO128F9322AE7 │ The Rolling Stones
│ Start Me Up
TRAERPT128F931103E │ The Rolling Stones
│ Time Is On My Side
TRAKBON128F9311039 │ The Rolling Stones
│ Around And Around
TRAHBWE128F9349247 │ The Shirelles
│ Dedicated To the One I Lov…
(33 rows)
```

The timing is key here, in terms of its order of magnitude. Using 10ms to search

your tags database leaves you with enough time on the frontend parts of your ap-

plication to keep your users happy, even when implementing advanced searches.

User-Defined Tags Made Easy

The usual way to handle a set of user-de ned tags and query against it involves

join against a reference table of tags, but then it's quite complicated to express the

full search query: we want tracks tagged with both *blu* and *rhythm and blu* , and might then want to exclude *finger picking*.

The intarray extension provides a powerful *query specialized language* with direct index support, so that you can build dynamic indexes searches directly from

your application.

47

The Most Popular Pub Names

PostgreSQL implements the *point* data type. Using this datatype, it's possible to register locations of points of interest on Earth, by using the point values as

coordinates for the longitude and latitude. The open source project Open Street

Map publishes geographic data that we can use, such as pubs in the UK.

A Pub Names Database

Using the Overpass API services and a URL like the following, we can download

an XML le containing geolocated pubs in the UK:

http://www.overpass-api.de/api/xapi?*[amenity=pub]

[bbox=-10.5,49.78,1.78,59]

The data itself is available from OSM in some kind of XML format where they

managed to handle the data in an EAV model:

```
<node id="262706" lat="51.0350300" lon="-0.7251785" >
<tag k="amenity" v="pub" />
<tag k="created_by" v="Potlatch 0.10f" />
<tag k="name" v="Kings Arms" />
</node>
```

In our context in this chapter, we only need a very simple database schema for

where to load this dataset, and the following is going to be ne for this purpose:

```
create table if not exists pubnames
(
id
bigint,
pos
point,
name text
);
```

So as to be able to load the data in a streaming fashion with the COPY protocol,

we are going to use a SAX API to read the XML. Here’s a slightly edited portion of the code I’ve been using to parse and load the data, available as the pubnames

project on *GitHub*. Once more, the script is written in Common Lisp:

```
(defun parse-osm-end-element (source stream)
"When we're done with a <node>, send the data over to the stream"
```

```
(when (and (eq 'node (current-qname-as-symbol source))
*current-osm*)
;; don't send data if we don't have a pub name
(when (osm-name *current-osm*)
(cl-postgres:db-write-row stream (osm-to-pgsql *current-osm*)))

;; reset *current-osm* for parsing the next <node>
(setf *current-osm* nil)))

(defmethod osm-to-pgsql ((o osm))
"Convert an OSM struct to a list that we can send over to PostgreSQL"
```

```
(list (osm-id o)
(format nil "(~a,~a)" (osm-lon o) (osm-lat o))
(osm-name o)))

(defun import-osm-file (&key
table-name sql pathname
(truncate t)
(drop nil))
"Parse the given PATHNAME file, formated as OSM XML."

(maybe-create-postgresql-table :table-name table-name
:sql sql
:drop drop
:truncate truncate)

(klacks:with-open-source (s (cxml:make-source pathname))
(loop
with stream =
(cl-postgres:open-db-writer (remove :port *pgconn*) table-name nil)
for key = (klacks:peek s)
while key
do
(case key
(:start-element (parse-osm-start-element s))
(:end-element
(parse-osm-end-element s stream)))
(klacks:consume s)


finally (return (cl-postgres:close-db-writer stream)))))
```

Given that code, we can parse the data in the XML le and load it into our Post-

greSQL table in a streaming fashion, using the PostgreSQL COPY protocol. We

use a SAX parser for the XML content, to which tag handler functions are regis-

tered:

• The parse-osm-start-element and parse-osm-end-element extract the information we need from the node and tag XML elements, and ll in our

OSM internal data structure.

• Once the node and tag XML elements are parsed into an OSM in-

memory structure, we serialize this record to PostgreSQL using the

cl-postgres:open-db-writer and osm-to-pgsql functions.

The Common Lisp driver for PostgreSQL that is used here is named Postmodern

and implements the COPY protocol with the three functions open-db-writer,

db-write-row, and close-db-writer, as we already saw earlier. Again, we're

using the COPY support from our PostgreSQL driver to stream the data as we

parse it.

It is of course possible to implement this approach in any programming lan-

guage.

Normalizing the Data

As we are interested in the most popular pub names in the United Kingdom, we

need to do some light data normalization. Of course, it's easy and e cient to do

that directly in SQL once the data has been loaded.

Here we're using the technique coined ELT rather than the more common ETL,

so extract, load, and only then transform the data:

select array_to_string(array_agg(distinct(name) order by name), ', '),

count(*)

from pubnames

group by replace(replace(name, 'The ', ''), 'And', '&')

order by count desc

limit 5;

In this query we group pub names that look alike. Here are then our most pop-

ular pub names, with their spelling alternatives, coma separated:

array_to_string

│ count

═══════════════════════════════════════╪═════════

Red Lion, The Red Lion

│

350

Royal Oak, The Royal Oak

│

287

Crown, The Crown

│

204

The White Hart, White Hart

|

180

The White Horse, White Horse |

163

(5 rows)

The array_to_string function allows us to tweak the output at our conve-

nience, as the array_agg(distinct(name) order by name) aggregate is doing

all the work for us here in grouping all *nam* together and keeping an ordered set of a unique entry per variant.

Which *nam* do we group together you might ask? Well, those having the same

name apart from some spelling variants: we don't want to consider The to be a

di ference so we replace it with an empty string, and we do want to consider both

and and & as the same thing too.

Geolocating the Nearest Pub (k-NN search)

To implement a k-NN search in PostgreSQL, we need to order the result set with a *distance* operator, written <->. Here's the full SQL for searching the pubs nearby a known position:

1

```
select id, name, pos
from pubnames
order by pos <-> point(-0.12,51.516)
limit 3;
```

With this geolocation, we obtain the following nearby pubs:

```
id
│
name
│
pos
═══════════╪═══════════════════════╪═══════
21593238 │ All Bar One
│ (-0.1192746,51.5163499)
26848690 │ The Shakespeare's Head │ (-0.1194731,51.5167871)
371049718 │ The Newton Arms
│ (-0.1209811,51.5163032)
(3 rows)
```

The PostgreSQL point datatype data type implement abstract coordinates in a two dimensional system, and it isn’t bound to any speci c projection of the

Earth. As a result, the distance operator is the Euclidian distance, and the *point* data type doesn’t implement Earth distance in meters or miles itself. There’s

more about that in the next example though, using the earthdistance exten-

sion.

Indexing kNN Search

The previous query ran in about 20ms. With a dataset of 27878 rows having an

answer in about 20ms is not a great achievement. Indeed, we didn’t create any

indexing whatsoever on the table yet, so the query planner has no other choice

but to scan the whole content on disk and lter it as it goes.

It would be much better for performance if we could instead evaluate our query

constraints (here, the ORDER BY and LIMIT clauses) using some index search in-

stead.

That’s exactly the kind of situation that GiST and SP GiST indexes have been designed to be able to solve for you in PostgreSQL, and in particular the kNN

GiST support. Let’s give it a try:

```
create index on pubnames using gist(pos);
```

With that index, we can now run the same query again, and of course we get the same result:

```
\pset format wrapped
\pset columns 72

explain (analyze, verbose, buffers, costs off)
select id, name, pos
from pubnames
order by pos <-> point(51.516,-0.12)
limit 3;
```

Here’s the query explain plan:

```
QUERY PLAN
══════════════════════════════════════════════════════════════════════
Limit (actual time=0.071..0.077 rows=3 loops=1)
Output: id, name, pos, ((pos <-> '(51.516,-0.12)'::point))
Buffers: shared hit=6
->
Index Scan using pubnames_pos_idx on public.pubnames (actual tim…
…e=0.070..0.076 rows=3 loops=1)
Output: id, name, pos, (pos <-> '(51.516,-0.12)'::point)
Order By: (pubnames.pos <-> '(51.516,-0.12)'::point)
Buffers: shared hit=6
Planning time: 0.095 ms
Execution time: 0.125 ms
```

There we go! With a dataset of 27878 rows in total, nding the three nearest

pubs in less than a millisecond is something we can actually be happy with, and

we can use this directly in a web application. I would expect this performance to

remain in the right ballpark even for a much larger dataset, and I’ll leave it as an exercise for you to nd that dataset and test the kNN GiST indexes on it!

How far is the nearest pub?

Computing the distance between two given positions on the Earth expressed as

longitude and *latitude* is not that easy. It involves knowing how to process the Earth as a sphere, and some knowledge of the projection system in which the

coordinates are valid. PostgreSQL makes it easy to solve though, thanks to the

earthdistance extension, included in contribs.

The earthdistance PostgreSQL contrib

As the mathematics are complex enough to easily make mistakes when imple-

menting them again, we want to nd an existing implementation that's already

been tested. PostgreSQL provides several contrib extensions: one of them is named earthdistance and it is made to solve our problem. Time to try it!

```
create extension cube;
create extension earthdistance;
```

Equipped with that extension we can now use its <@> operator and compute a distance in miles on the surface of the Earth, given points as *(longitude, latitude)*:

```
select id, name, pos,
       round((pos <@> point(-0.12,51.516))::numeric, 3) as miles
  from pubnames
order by pos <-> point(-0.12,51.516)
   limit 10;
```

We now have our ten closests pubs, and the distance to get there in miles!

```
id │ name │ pos │ miles
═══════════╪═══════════════════════╪═══════
21593238 │ All Bar One
```

```
│ (-0.1192746,51.5163499) │ 0.039

26848690 │ The Shakespeare's Head │ (-0.1194731,51.5167871) │ 0.059

371049718 │ The Newton Arms

│ (-0.1209811,51.5163032) │ 0.047

438488621 │ Marquis Cornwallis

│ (-0.1199612,51.5146691) │ 0.092

21593236 │ Ship Tavern

│ (-0.1192378,51.5172525) │ 0.093

312156665 │ The Prince of Wales

│ (-0.121732,51.5145794)

│ 0.123

312156722 │ O'Neills

│ (-0.1220195,51.5149538) │ 0.113

25508632 │ Friend at Hand

│ (-0.1224717,51.5148694) │ 0.132

338507304 │ The Square Pig

│ (-0.1191744,51.5187089) │ 0.191

1975855516 │ Holborn Whippet

│ (-0.1216925,51.5185189) │ 0.189

(10 rows)
```

So the nearest pub is *All Bar One*, 0.039 miles away, or apparently 68.64 yards.

Also, adding the computation to get the distance in *mil* didn’t add that much to the query timing, which remains well under a millisecond ona laptop when

data is available in memory.

Pubs and Cities

Just as easily as we found the *nearest* pubs we can also of course query for the pubs that are *farthest* away from any location.

```
select name, round((pos <@> point(-0.12,51.516))::numeric, 3) as miles
from pubnames
order by pos <-> point(-0.12,51.516) desc
limit 5;
```

I’m not sure how useful that particular query would be. That said, it shows that

the kNN search supports the ORDER BY DESC variant:

```
name | miles
═════╪══════
```

```
Tig Bhric

 │ 440.194

TP's

 │ 439.779

Begley's

 │ 439.752

Ventry Inn

 │ 438.962

Fisherman's Bar │ 439.153

(5 rows)
```

Now we want to know what city those pubs are in, right? With the following

URL and using the Open Street Map APIs, I’ve been able to download a list of

cities in the same area as where the pub names were fetched:

```
http://www.overpass-api.de/api/xapi?*[place=city]
[bbox=-10.5,49.78,1.78,59]
```

Tweaking the parser and import code at https://github.com/dimitri/pubnames

was easy, and allowed to import those city names and locations in *0.087 seconds*

of real time, with the following schema:

```
create table if not exists cities
(
id
bigint,
pos point,
name text
);

create index on cities using gist(pos);
```

Now let’s see where those far away pubs are:

```
select name,
(select name from cities c order by c.pos <-> p.pos limit 1) as city,
round((pos <@> point(-0.12,51.516))::numeric, 3) as miles
from pubnames p
order by pos <-> point(-0.12,51.516) desc
limit 5;
```

This time, we get the name of the closest known city to the pub:

```
name
│
city
│
miles
════════════════════╪══════════╪══════════
Tig Bhric
│ Galway │ 440.194
TP's
│ Galway │ 439.779
Begley's
```

```
│ Galway │ 439.752
Ventry Inn
│ Galway │ 438.962
Fisherman's Bar │ Cork
│ 439.153
(5 rows)
```

As you can see we are fetching the pubs at a distance from our given point and then the nearest city to the location of the pub. The way it’s implemented here is called a *correlated subquery*.

It’s also possible to write such a query as a LATERAL join construct, as in the following example:

```
select c.name as city, p.name,
round((pos <@> point(-0.12,51.516))::numeric, 3) as miles
from pubnames p,
lateral (select name
from cities c
order by c.pos <-> p.pos
limit 1) c
```

```
order by pos <-> point(-0.12,51.516) desc
limit 5;
```

It provides the same result of course:

```
city
│
name
│
miles
═══════════╪════════════════════════╪════════════
Galway │ Tig Bhric
│ 440.194
```

```
Galway │ TP's

│ 439.779

Galway │ Begley's

│ 439.752

Galway │ Ventry Inn

│ 438.962

Cork

│ Fisherman's Bar │ 439.153

(5 rows)
```

So apparently the *bounded box* that we’ve been given ([bbox=-10.5,49.78,1.78,59]) includes Ireland too… and more importantly the query execution penalty is

quite important.

That’s because the planner only know how to solve that query by scanning the

position index of the cities 27878 times in a loop (once per pubnames entry), as

we can see in this explain (analyze, costs off) output:

```
QUERY PLAN

═══════════════════════════════════════════════════════════════

Limit (actual time=1323.517..1323.518 rows=5 loops=1)

->
```

```
Sort (actual time=1323.515..1323.515 rows=5 loops=1)

Sort Key: ((p.pos <-> '(-0.12,51.516)'::point)) DESC

Sort Method: top-N heapsort

Memory: 25kB

->

Nested Loop (actual time=0.116..1310.214 rows=27878 loops=…

…1)

->

Seq Scan on pubnames p (actual time=0.015..4.465 row…

…s=27878 loops=1)

->

Limit (actual time=0.044..0.044 rows=1 loops=27878)

->

Sort (actual time=0.043..0.043 rows=1 loops=27…

…878)

Sort Key: ((c.pos <-> p.pos))

Sort Method: top-N heapsort

Memory: 25kB

->

Seq Scan on cities c (actual time=0.003.…
….0.019 rows=73 loops=27878)
Planning time: 0.236 ms
Execution time: 1323.592 ms
(13 rows)
```

It’s possible to force the planner into doing it the obvious way though:

```
with pubs as (
select name, pos,
round((pos <@> point(-0.12,51.516))::numeric, 3) as miles
from pubnames
order by pos <-> point(-0.12,51.516) desc
limit 5
)
```

```
select c.name as city, p.name, p.miles
  from pubs p, lateral (select name
                          from cities c
                      order by c.pos <-> p.pos
                         limit 1) c;
```

We still get the same result of course, this time in about 60ms rather than more than a second as happened before:

```
 city   │   name    │  miles
════════╪═══════════╪═════════
 Galway │ Tig Bhric │ 440.194
Galway │ TP's
│ 439.779
Galway │ Begley's
│ 439.752
Galway │ Ventry Inn
│ 438.962
Cork
│ Fisherman's Bar │ 439.153
(5 rows)
```

The Most Popular Pub Names by City

Let’s now nd out which cities have the highest count of pubs, considering that

a pub is a liated with a city if it’s within ve miles of the single point we have as

city location in our data set.

```
select c.name, count(cp)
from cities c, lateral (select name
from pubnames p
where (p.pos <@> c.pos) < 5) as cp
group by c.name
order by count(cp) desc
limit 10;
```

We use that method of associating pubs and cities because within the data we exported from Open Street Map, the only information we have is a single point to represent a city. So our method amounts to drawing a 5-mile circle around that point, and then consider that anything that’s inside the circle to be part of the town.

```
name
 │ count
═══════════════════╪══════════
London
 │
1388
Westminster │
```

1383

Dublin

|

402

Manchester

|

306

Bristol

|

292

Leeds

|

292

Edinburgh

|

286

Liverpool

|

258

Nottingham

|

218

Glasgow

|

217

(10 rows)

If we look at a map we see that *Westminster* is in fact within *London* given our arbitrary rule of *within 5 mil* , so in the next query we will simply lter it out.

Exercise lef to the reader: write a query allowing to remove from

London’s count the pubs that are actually in Westminster (when

within 1 mile of the location we have for it). Then extend that query

to address any other situation like that in the whole data set.

It’s a good time to hint towards using PostGIS here if your application needs to consider the real world shapes of cities rather than

playing *guestimat* as we are doing here.

And now what about the most popular pub names per city? Of course we want

to normalize our pub names again here but only for counting: we still display all

the names we did count.

```
select c.name,
array_to_string(array_agg(distinct(cp.name) order by cp.name), ', '),
count(*)
from cities c,
lateral (select name
from pubnames p
where (p.pos <@> c.pos) < 5) as cp
where c.name <> 'Westminster'
group by c.name, replace(replace(cp.name, 'The ', ''), 'And', '&')
order by count(*) desc
```

```
limit 10;
```

This query uses all the previous tricks:

• A lateral subquery

• Data normalization done within the query

• Distance computations thanks to the <@> point operator provided by the earthdistance extension

• We add an ordered aggregate that removes duplicates

In case you might be curious, here’s the result we get:

```
name
│
array_to_string
│ count
══════════════╪══════════════════════════════════════════════
```

```
London
│ Prince of Wales, The Prince of Wales
│
15
London
│ All Bar One
```

|

12

London

| The Beehive

|

8

London

| O'Neills

|

7

London

| The Crown

|

7

London

| The Windmill

|

7

London

| Coach and Horses, The Coach and Horses |

```
6

London

│ The Ship

│

6

Bradford │ New Inn, The New Inn

│

6

London

│ Red Lion, The Red Lion

│

6

(10 rows)
```

Geolocation with PostgreSQL

We have loaded Open Street Map points of interests in the previous section: a

localized set of pubs from the UK. In this section, we are going to have a
look at

how to geolocalize an IP address and locate the nearest pub, all within a
single

SQL query!

For that, we are going to use the awesome ip4r extension from RhodiumToad.

Geolocation Data Loading

The rst step is to nd an *geolocation* database, and several providers o fer that.

The one I ended up choosing for the example is the http://www.maxmind.com

free database available at GeoLite Free Downloadable Databases.

Af er having had a look at the les there, we de ne the table schema we want and

load the archive using pgloader. So, rst, the target schema is created using the

following script:

```
create extension if not exists ip4r;
create schema if not exists geolite;

create table if not exists geolite.location
(
```

locid

integer **primary key**,

7

country

text,

8

region

text,

9

city

text,

10

postalcode text,

11

location

point,

12

metrocode

text,

```
areacode text
);

create table if not exists geolite.blocks
(
iprange ip4r,
locid integer
);

```

create index blocks_ip4r_idx **on** geolite.blocks **using** gist(iprange); The data can now be imported to those target tables thanks to the following

pgloader command, which is quite involved:

/*

* Loading from a ZIP archive containing CSV files.

*/

LOAD ARCHIVE

FROM http://geolite.maxmind.com/download/geoip/database/GeoLiteCity_CSV/Geo latest.zip INTO postgresql://appdev@/appdev

BEFORE LOAD EXECUTE 'geolite.sql'

LOAD CSV

FROM FILENAME MATCHING ~/GeoLiteCity-Location.csv/

WITH ENCODING iso-8859-1

(

locId,

country,

region

[null if blanks],

city

[null if blanks],

```
postalCode [ null if blanks ],

latitude,

longitude,

metroCode

[ null if blanks ],

areaCode

[ null if blanks ]

)

INTO postgresql://appdev@/appdev

TARGET TABLE geolite.location

(

locid,country,region,city,postalCode,

location point using (format nil "(~a,~a)" longitude latitude),

metroCode,areaCode

)

WITH skip header = 2,

drop indexes,

fields optionally enclosed by '"',

fields escaped by double-quote,

fields terminated by ','
```

```
AND LOAD CSV
```

```
FROM FILENAME MATCHING ~/GeoLiteCity-Blocks.csv/
WITH ENCODING iso-8859-1
(
startIpNum, endIpNum, locId
)
INTO postgresql://appdev@/appdev
TARGET TABLE geolite.blocks
(
iprange ip4r using (ip-range startIpNum endIpNum),
locId
)
WITH skip header = 2,
drop indexes,
fields optionally enclosed by '"',
fields escaped by double-quote,
fields terminated by ',';
```

The pgloader command describe the le format so that pgloader can parse the data from the CSV le and transform it in memory to the format we expect in

PostgreSQL. The location in the CSV le is given as two separate elds latitude and longitude, which we use to form a single point column.

In the same vein, the pgloader command also describes how to transform an IP address range from a couple of integers to a more classic representation of the same data:

```
CL-USER> (pgloader.transforms::ip-range "16777216" "16777471")
"1.0.0.0-1.0.0.255"
```

The pgloader command also nds the les we want to load independently from the real name of the directory, here GeoLiteCity_20180327. So when there’s a new release of the *Geolite* les, you can run the pgloader once again and expect it to load the new data.

Here’s what the output of the pgloader command looks like. Note that I have stripped the timestamps from the logs output, in order for the line to t in those pages:

```
$ pgloader --verbose geolite.load
NOTICE Starting pgloader, log system is ready.
```

```
LOG Data errors in '/private/tmp/pgloader/'

LOG Parsing commands from file #P"/Users/dim/dev/yesql/src/1-
application-development/data/geolite/geolite.load"

LOG Fetching
'http://geolite.maxmind.com/download/geoip/database/GeoLiteCity_CSV/Geo
latest.zip'

LOG Extracting files from archive
'/var/folders/bh/t1wcr6cx37v4h87yj3qj009r0000gn/T/GeoLiteCity-latest.zip'

NOTICE unzip -o
"/var/folders/bh/t1wcr6cx37v4h87yj3qj009r0000gn/T/GeoLiteCity-latest.zip"
-d "/var/folders/bh/t1wcr6cx37v4h87yj3qj009r0000gn/T/GeoLiteCity-latest/"

NOTICE Executing SQL block for before load

NOTICE ALTER TABLE "geolite"."location" DROP CONSTRAINT IF
EXISTS "location_pkey" ;

NOTICE COPY "geolite"."location"
```

NOTICE Opening *#P"/private/var/folders/bh/t1wcr6cx37v4h87yj3qj009r0000gn/T/GeoLiteCity-latest/GeoLiteCity_20180327/GeoLiteCity-Location.csv"*

12

NOTICE copy "geolite"."location": 234105 rows done,

11.5 MB,

2.1 MBps

13

NOTICE copy "geolite"."location": 495453 rows done,

24.3 MB,

2.2 MBps

14

NOTICE copy "geolite"."location": 747550 rows done,

37.1 MB,

2.2 MBps

15

NOTICE CREATE UNIQUE INDEX location_pkey ON geolite.location USING btree (locid)

16

NOTICE ALTER TABLE "geolite"."location" ADD PRIMARY KEY USING INDEX "location_pkey" **;** 17

```
NOTICE DROP INDEX IF EXISTS "geolite"."blocks_ip4r_idx" ;
NOTICE COPY "geolite"."blocks"
NOTICE Opening
#P"/private/var/folders/bh/t1wcr6cx37v4h87yj3qj009r0000gn/T/GeoLiteCity-
latest/GeoLiteCity_20180327/GeoLiteCity-Blocks.csv"
NOTICE copy "geolite"."blocks": 227502 rows done,
7.0 MB,
1.8 MBps
NOTICE copy "geolite"."blocks": 492894 rows done,
15.2 MB,
1.9 MBps
NOTICE copy "geolite"."blocks": 738483 rows done,
22.9 MB,
2.0 MBps
NOTICE copy "geolite"."blocks": 986719 rows done,
30.7 MB,
2.1 MBps
NOTICE copy "geolite"."blocks": 1246450 rows done,
38.9 MB,
2.2 MBps
NOTICE copy "geolite"."blocks": 1489726 rows done,
47.1 MB,
2.2 MBps
NOTICE copy "geolite"."blocks": 1733633 rows done,
55.1 MB,
2.2 MBps
NOTICE copy "geolite"."blocks": 1985222 rows done,
63.3 MB,
2.2 MBps
NOTICE CREATE INDEX blocks_ip4r_idx ON geolite.blocks USING gist
(iprange)
```

29

LOG report summary reset

30

table name

errors

read

imported

bytes

total time

31

32

download

0

0

0

0.793s

33

extract

0

0

0

0.855s

34

before load

0

5

5

0.033s

35

fetch

0

0

0

0.005s

```
36

-----------------------

---------

---------

---------

---------

--------------

37

"geolite"."location"

0

928138

928138

46.4 MB

20.983s

38

"geolite"."blocks"

0

2108310

2108310

67.4 MB
```

```
30.695s

39

-----------------------

---------

---------

---------

---------

--------------

40

Files Processed

0

2

2

0.024s

41

COPY Threads Completion

0

4

4

51.690s
```

42

Index Build Completion

0

0

0

49.363s

43

Create Indexes

0

2

2

49.265s

44

Constraints

0

1

1

0.002s

45

46

Total import time

✓

3036448

3036448

113.8 MB

2m30.344s

We can see that pgloader has dropped the indexes before loading the data, and

created them again once the data is loaded, in parallel to loading data from the

next table. This parallel processing can be a huge bene t on beefy servers.

So we now have the following tables to play with:

List of relations

Schema

|

```
Name
│ Type
│ Owner
│ Size
│ Description
═════════════╪═══════════════╪═══════════╪════════════╪═════════
geolite │ blocks
│ table │ appdev │ 89 MB │
geolite │ location │ table │ appdev │ 64 MB │
(2 rows)
```

Finding an IP Address in the Ranges

Here’s what the main data looks like:

```
table geolite.blocks limit 10;
```

The TABLE command is SQL standard, so we might as well use it:

```
iprange
│ locid
══════════════════════════════╪════════════
1.0.0.0/24
```

│ 617943

1.0.1.0-1.0.3.255

│ 104084

1.0.4.0/22

│

17

1.0.8.0/21

│

47667

1.0.16.0/20

│ 879228

1.0.32.0/19

│

47667

1.0.64.0-1.0.81.255 │ 885221

1.0.82.0/24

│ 902132

1.0.83.0-1.0.86.255 │ 885221

1.0.87.0/24

│ 873145

(10 rows)

What we have here is an *ip range* column. We can see that the output function for ip4r is smart enough to display ranges either in their CIDR notation or in the more general *start-end* notation when no CIDR applies.

The *ip4r* extension provides several operators to work with the dataset we have, and some of those operators are supported by the index we just created. Just for

the fun of it here’s a catalog query to inquire about them:

```
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 catalog query above joins the PostgreSQL catalogs for operator classes, and

uses index access methods according to the notion of an operator family in order

to retrieve the list of operators associated with the ip4r data type and the

```
GiST
access method:
amopopr
══════════════════════
>>=(ip4r,ip4r)
<<=(ip4r,ip4r)
>>(ip4r,ip4r)
<<(ip4r,ip4r)
&&(ip4r,ip4r)
=(ip4r,ip4r)
```

```
(6 rows)
```

Note that we clearly could have been using the psql \dx+ ip4r command instead, but that query directly list operators that the *GiST* index knows how to solve. The operator >>= reads as *contains* and is the one we're going to use here.

```
select iprange, locid
from geolite.blocks
```

where iprange >>= '91.121.37.122';

So here’s the range in which we nd the IP address 91.121.37.122, and the location

it’s associated with:

iprange

│ locid

═══════════════════════════════════╪══════════

91.121.0.0-91.121.71.255 │

75

This lookup is fast, thanks to our specialized GiST index. Its timing is under a

millisecond.

Geolocation Metadata

Now with the *MaxMind* schema that we are using in that example, the inter-

esting data is actually to be found in the other table, i.e. geolite.location.

Let’s use another IP address now — I’m told by the unix command host that

google.us has address 74.125.195.147 and we can inquire where that IP is

from:

1

select *

2

from

geolite.blocks

3

join geolite.location **using**(locid)

4

where iprange >>= '74.125.195.147';

Our data locates the Google IP address in Mountain View, which is credible:

─[RECORD 1]───────────────────────────────

locid

│ 2703

iprange

│ 74.125.191.0-74.125.223.255

country

│ US

region

│ CA

city

│ Mountain View

postalcode │ 94043

location

```
│ (-122.0574,37.4192)
metrocode
│ 807
areacode
│ 650
```

Now you can actually draw that on a map as you have the location information

as a *point* datatype containing both the *longitude* and *latitude*.

Emergency Pub

What if you want to make an application to help lost souls nd the nearest pub

from where they are currently? Now that you know their location from the *IP*

address they are using in their browser, it should be easy enough right?

As we downloaded a list of pubs from the UK, we are going to use an IP address

that should be in the UK too:

```
$ host www.ox.ac.uk
www.ox.ac.uk has address 129.67.242.154
www.ox.ac.uk has address 129.67.242.155
```

Knowing that, we can search the geolocation of this IP address:

```
select *
from
geolite.location l
join geolite.blocks using(locid)
where iprange >>= '129.67.242.154';
```

And the Oxford University is actually hosted in Oxford, it seems:

```
─[ RECORD 1 ]───────────────────
locid
│ 375290
country
│ GB
region
│ K2
city
```

```
│ Oxford
postalcode │ OX1
location
│ (-1.25,51.75)
metrocode
│ ¤
areacode
│ ¤
iprange
│ 129.67.0.0/16
```

What are the ten nearest pubs around you if you’re just stepping out of the Oxford University? Well, let’s gure that out before we get too thirsty!

```
select pubs.name,
round((pubs.pos <@> l.location)::numeric, 3) as miles,
ceil(1609.34 * (pubs.pos <@> l.location)::numeric) as meters


```

```
from geolite.location l
join geolite.blocks using(locid)
left join lateral
```

```
(
select name, pos
from pubnames
order by pos <-> l.location
limit 10
) as pubs on true
```

where blocks.iprange >>= '129.67.242.154'

16

order by meters;

Here’s the list, obtained in around about a millisecond on my laptop:

```
name

│ miles │ meters

═════════════════════════════╪═══════════╪════════════

The Bear

│ 0.268 │

431

The Half Moon

│ 0.280 │

451

The Wheatsheaf

│ 0.295 │

475

The Chequers

│ 0.314 │

506

The Old Tom
```

```
│ 0.315 │

507

Turl Bar

│ 0.321 │

518

St Aldate's Tavern │ 0.329 │

530

The Mad Hatter

│ 0.337 │

542

King's Arms

│ 0.397 │

639

White Horse

│ 0.402 │

647

(10 rows)
```

So with PostgreSQL and some easily available extensions, we are actually capable

of performing advanced geolocation lookups in a single SQL query. In addition, with query timing between 1ms and 6ms, it is possible to use this technique in

production, serving users requests directly from the live query!

50

Counting Distinct Users with

HyperLogLog

If you’ve been following along at home and keeping up with the newer statistics

developments, you might have heard about this new state of the art cardinality

estimation algorithm called HyperLogLog.

This technique is now available for PostgreSQL in the extension postgresql-hll

available at https://github.com/citusdata/postgresql-hll and is packaged for multiple operating systems such as Debian and RHEL, through the PostgreSQL

community packaging e forts and resources.

HyperLogLog

HyperLogLog is a very special hash value. It aggregates enough information into

a single scalar value to compute a distinct value with some precision loss.

Say we are counting unique visitors. With HyperLogLog we can maintain a sin-

gle value per day, and then *union* those values together to obtain unique weekly or monthly visitor counts!

Here’s an example in SQL of the magic provided by the hll extension:

```
select to_char(date, 'YYYY/MM') as month,
round(#hll_union_agg(users)) as monthly
```

```
from daily_uniques
group by month;
```

While we are keeping daily aggregates on disk, we can use the HyperLogLog maths to *union* them together and compute an approximation of the monthly unique count from the same dataset!

```
month
| monthly
---------+---------
2013/02 | 1960380
(1 row)
```

So by keeping only a small amount of data per day, typically 1280 bytes, it is then possible to compute monthly unique counts from that, without having to scan

a whole month of records again.

Installing postgresql-hll

It’s as simple as create extension hll;, once the OS package is installed on

your system. The extension provides a new datatype named hll and we can use

\dx+ hll to discover what kind of support comes with it. Here’s an edited ver-

sion of the output of the \dx+ hll command, where some lines have been l-

tered out of the 71 SQL objects:

Objects in extension "hll"

Object description

---

cast from bigint to hll_hashval

cast from bytea to hll

cast from hll to hll

cast from integer to hll_hashval

function hll(hll,integer,boolean)

function hll_add(hll,hll_hashval)

function hll_add_agg(hll_hashval)

function hll_add_agg(hll_hashval,integer)

function hll_add_agg(hll_hashval,integer,integer)

```
function hll_add_agg(hll_hashval,integer,integer,bigint)

function hll_add_agg(hll_hashval,integer,integer,bigint,integer)

function hll_add_rev(hll_hashval,hll)

...

function hll_cardinality(hll)

function hll_empty()

function hll_eq(hll,hll)

...

function hll_hash_any(anyelement,integer)

function hll_hash_bigint(bigint,integer)

function hll_hash_boolean(boolean,integer)

function hll_hash_bytea(bytea,integer)
```

```
function hll_hash_integer(integer,integer)

function hll_hash_smallint(smallint,integer)

function hll_hash_text(text,integer)

...

operator #(NONE,hll)

operator <>(hll,hll)

operator <>(hll_hashval,hll_hashval)
```

operator =(hll,hll)

operator =(hll_hashval,hll_hashval)

operator ||(hll,hll)

operator ||(hll,hll_hashval)

operator ||(hll_hashval,hll)

type hll

type hll_hashval

From that output we learn the list of hll operators, such as the interesting #

operator, a unary operator that works on an hll value. More about this one

later…

Counting Unique Tweet Visitors

As an example use case for the HyperLogLog data type, we are going to count

unique visitors to our tweets, using the application we introduced in Data Ma-

nipulation and Concurrency Control.

The two main operations around an hll data type consists of the following:

• Build a hash from an input value, such as an IP address.

• Update the already known hll value with the hash.

The main idea behind hll is to keep a single hll value per *granularity*, here per tweet message and per day. This means that each time we have a new visit on a tweet, we want to UPDATE our hll set to count that visitor.

As we have seen in the previous chapter, concurrency is a deal breaker for UP-

DATE heavy scenarios where the same row is solicited over and over again. So

we are going to work in two steps again here, rst doing an INSERT per visit and

then arranging a background process to transform those visits into an UPDATE

to the single hll aggregate per tweet and date.

Here’s the visitor table where we can insert every single visit:

```
create table tweet.visitor
(
id
bigserial primary key,
```

```
messageid
bigint not null references tweet.message(messageid),
datetime timestamptz not null default now(),
ipaddr ipaddress,

unique(messageid, datetime, ipaddr)
);
```

It’s a pretty simple structure, and is meant to register our online activity.

We can generate some tweet visits easily with a program such as the following.

Again, I’m using Common Lisp to implement a very simple COPY-based load-
ing program.

```
(defparameter *connspec* '("appdev" "dim" nil "localhost"))
(defparameter *visitor-table* "tweet.visitor")
```

```
(defparameter *visitor-columns* '("messageid" "ipaddr" "datetime"))

(defun insert-visistors (messageid n &optional (connspec *connspec*))
(pomo:with-connection connspec
(let ((count 0)
(copier (open-db-writer connspec *visitor-table* *visitor-columns*)))
(unwind-protect
(loop :for i :below n
:do (let ((ipaddr
(generate-ipaddress))
(datetime (format nil "~a" (generate-timestamp))))

(db-write-row copier (list messageid ipaddr datetime))
(incf count)))
(close-db-writer copier))

;; and return the number of rows copied
count)))
```

The script is written so as to target a *smallish* range of IP addresses and range of dates in order to generate collisions and see our unique visitors count as being

more than one.

```
;;;
;;; select '192.168.0.0'::ip4::bigint; == 3232235520
;;;
```

```
(defparameter *ip-range-start* 3232235520)
(defparameter *ip-range-size* (expt 2 16))

(defun generate-ipaddress (&optional
(range-size *ip-range-size*)
(range-start *ip-range-start*))
"Generate N random IP addresses, as strings."
(int-to-ip (+ range-start (random range-size))))

(defun generate-timestamp ()
"Generate a random timestamp between now and a month ago."
(local-time:timestamp- (local-time:now) (random #. (* 24 60 31)) :minute))
```

When generating data with those function, we pick the subnet in 192.168.0.0/16

and a span of a month of data. Here’s how to interactively generate 100 000

visits from the Common Lisp REPL, measuring the time that takes:

```
CL-USER> (time (shakes::insert-visistors 3 100000))
(SHAKES::INSERT-VISISTORS 3 100000)
took 7,513,209 microseconds (7.513209 seconds) to run.
244,590 microseconds (0.244590 seconds, 3.26%) of which was spent in GC.
During that period, and with 4 available CPU cores,
5,242,334 microseconds (5.242334 seconds) were spent in user mode
314,728 microseconds (0.314728 seconds) were spent in system mode
```

```
691,153,296 bytes of memory allocated.
770 minor page faults, 0 major page faults, 0 swaps.
100000
```

Thanks to using the COPY streaming protocol, we can mix generating the numbers and communicating with the PostgreSQL server, and have our hundred thousand visits be generated in the database in less than 8s on my laptop. That’s certainly fast enough for interactive discovery of a data model. It’s quite easy with PostgreSQL to just *try it and see*.

We can check the result of inserting 100000 visits to the messageid 3 with the following query:

```
select messageid,
datetime::date as date,
count(*) as count,
count(distinct ipaddr) as uniques,
count(*) - count(distinct ipaddr) as duplicates
from tweet.visitor
where messageid = 3
group by messageid, date
order by messageid, date
limit 10;
```

We have a precise count of all the visitors to the message, and we can see that even with a 16-bits range of IP addresses we already have several visits from the same IP addresses.

```
messageid │
date
│ count │ uniques │ duplicates
```

3 │ 2018-08-07 │

746 │

742 │

4

3 │ 2018-08-08 │

3298 │

3211 │

87

3 │ 2018-08-09 │

3260 │

3191 │

69

3 │ 2018-08-10 │

3156 │

3077 │

79

3 │ 2018-08-11 │

3241 │

3161 │

```
80
3 │ 2018-08-12 │
3270 │
3197 │
73
3 │ 2018-08-13 │
3182 │
3106 │
76
3 │ 2018-08-14 │
3199 │
3124 │
75
```

```
3 │ 2018-08-15 │
3308 │
3227 │
81
3 │ 2018-08-16 │
3261 │
```

```
3184 │
77
(10 rows)
```

Lossy Unique Count with HLL

We can rewrite the previous query using our HLL data type now, even though

at this stage it’s not going to be very useful, because we still have the full logs of every visit and we can a ford to compute precise counts.

Nonetheless, our goal is to dispose of the daily entries, that we anticipate will be

just too large a data set. So, the hll-based query looks like this:

```
select messageid,
datetime::date as date,
# hll_add_agg(hll_hash_text(ipaddr::text)) as hll
from tweet.visitor
where messageid = 3
```

```
group by grouping sets((messageid),
(messageid, date))
order by messageid, date nulls first
limit 10;
```

In this query we use several new functions and operators related to the hll data

type:

• The # operator takes a single argument: it’s a unary operator, like factorial

(written !) for example. This unary operator when applied to a value of

type hll computes the estimated number of distinct entries *stored* in the

hyperloglog set.

• The hll_add_agg() aggregate function accumulates new hashes into a

given hyperloglog set.

• The hll_hash_text function computes the hyperloglog hash of a text

value, here used with the IP address as a text form. We could also use

the IP address as a 32-bit integer with the hll_hash_integer function

instead, but then this wouldn’t support IPv6 addresses, which only t in a 128-bit number.

The notation # hll shows the level of exibility that PostgreSQL brings to the table with its extensibility support. Not only can you de ne new operators at runtime from an extension, but those operators can also be *unary* or *binary*.

The lossy distinct count result looks like this:

```
messageid │
date
│
hll
═══════════════╪═════════════════╪════════════════════════
3 │ 2018-08-07 │ 739.920627061887
3 │ 2018-08-08 │ 3284.16386418662
3 │ 2018-08-09 │ 3196.58757626223
3 │ 2018-08-10 │ 3036.32707701154
3 │ 2018-08-11 │ 3140.21704515932
3 │ 2018-08-12 │ 3191.83031512197
3 │ 2018-08-13 │ 3045.15467688584
3 │ 2018-08-14 │ 3031.92750496513
3 │ 2018-08-15 │ 3135.58879460201
```

```
3 │ 2018-08-16 │ 3230.20146096767
(10 rows)
```

When used that way, the hll feature set doesn’t make much sense. We still have

to process as many rows as before, but we lose some precision in the result. The

reason why we’ve done that query here is to show the following:

1. Demonstrate how to use the hll operators and functions in a query

2. Show that the estimates from the hll data structures are pretty good, even

at this low cardinality

Getting the Visits into Unique Counts

In a production setup we would have the following context and constraints:

• Tweets are published and users from the Internet are visiting our tweets.

• Our application inserts a new row in tweet.visitor with the visitor’s IP

address each time there is a new visit to one of our tweet. It also registers

the precise timestamp of the visit.

• As we anticipate quite some success on our little application idea, we also

anticipate not being able to keep all the visitor logs, and not being able to

respect our quality of service terms when computing the unique visitors

on the y each time someone needs them.

• Finally, as the numbers being used in a marketing context rather than in an invoicing context, we are in a position to lose some precision over the number, and we would actually like to implement a system that is lossy if it allows us to relax our storage and processing requirements.

The previous sections present a great tool for achieving the last point above, and now is the time to put hll to good use. From the tweet.visitor table we are now going to compute a single hyperloglog value per message and per day:

```
begin;

with new_visitors as
(
delete from tweet.visitor
where id = any (
select id
from tweet.visitor
order by datetime, messageid
for update
skip locked
limit 1000
)
returning messageid,
cast(datetime as date) as date,
hll_hash_text(ipaddr::text) as visitors
),
new_visitor_groups as
(
select messageid, date, hll_add_agg(visitors) as visitors
from new_visitors
group by messageid, date
)
insert into tweet.uniques
select messageid, date, visitors
from new_visitor_groups
on conflict (messageid, date)
```

```
do update set visitors = hll_union(uniques.visitors, excluded.visitors)
where uniques.messageid = excluded.messageid
and uniques.date = excluded.date
returning messageid, date, # visitors as uniques;

rollback;
```

This query is implemented in several stages thanks to the PostgreSQL support for *writable common table expressions*:

1. Compute new_visitors by deleting from the *buffer* table tweet.visitor a thousand rows at a time, and using the skip locked facility that is new in PostgreSQL 9.5.

By default, when attempting to delete a row that is already in use by another transaction doing either an update or a delete, PostgreSQL would have to block until the other transaction released its lock. With the skip locked clause, PostgreSQL can omit the row from the current transaction without incurring any locking or waiting.

Rows skipped that way may appear in the next batch, or they may already be concurrently processed in another batch.

This construct allows the query to be run in more than one transaction at the same time, which might in turn be useful if we ever have some serious lag in our processing.

2. This rst CTE of our query then also computes the date from the timestamp with a CAST expression, and the hll hash from the IP address, preparing for the next stage of processing.

3. Compute the new_visitor_groups by aggregating the just computed hll individual hashes into a single hll set per messageid and per date.

4. Finally, insert those messages daily unique visitors hll sets into our summary table tweet.uniques. Of course, if we did compute a set for the same message and the same day before, we then update and *hll_union* the existing and the new set together.

5. Because PostgreSQL is such a powerful system, of course we return the result of processing the given batch at the end of the query, using the returning clause of the insert command.

The do update set clause requires that any single row in the target table be updated only once per command, in order to ensure that the con ict handling

mechanism is deterministic. That’s the reason why we prepare the hll sets in the

new_visitor_groups CTE part of the query.

When running this query, we obtain the following result:

```
BEGIN

messageid │

date

│

uniques

═══════════════╪════════════════╪═════════════════════════

3 │ 2018-08-07 │ 739.920627061887

3 │ 2018-08-08 │ 257.534468469694

(2 rows)

INSERT 0 2

ROLLBACK
```

Notice that we nish our script with a rollback command. That allows us to

debug and re ne the query until we’re happy. This 5-stage, 29-line SQL query

isn’t going to be too complex to maintain thanks to its actions being well sepa-

rated using CTE, it still doesn’t get written in a single session in a text le. It gets

brewed at your favorite SQL prompt and re ned until satisfactory, and it being

a DML query, we prefer to rollback and try again rather than impact the data

set and have to clean it up for the next iteration.

Scheduling Estimates Computations

Now that we know how to compute unique visitors approximations from the

insert heavy table, we need to have a background process that runs this processing

every once in a while.

The easiest way to do that here would be to create a new API endpoint on your

backend server and set up a cron-like utility to use that endpoint for your speci-

ed schedule. In case of emergency though, it’s nice to be able to run this updat-

ing process interactively. A solution to have both the backend API integration

and the interactive approaches available consist of *packaging* your SQL query as a stored procedure.

While stored procedures aren’t covered in this book, it’s easy enough to write a

SQL function around the statement we have already:

```
begin;

create function tweet.update_unique_visitors
(
in batch_size bigint default 1000,
out messageid bigint,
out date date,
out uniques bigint

```

```
)
returns setof record
language SQL
as $$
with new_visitors as
(
delete from tweet.visitor
where id = any (
select id
from tweet.visitor
```

```sql
order by datetime, messageid
for update
skip locked
limit update_unique_visitors.batch_size
)
returning messageid,
```

```sql
cast(datetime as date) as date,
hll_hash_text(ipaddr::text) as visitors
),
new_visitor_groups as
```

```
(
select messageid, date, hll_add_agg(visitors) as visitors
from new_visitors
group by messageid, date
)
insert into tweet.uniques
select messageid, date, visitors
from new_visitor_groups
on conflict (messageid, date)
do update set visitors = hll_union(uniques.visitors, excluded.visitors)
```

```
where uniques.messageid = excluded.messageid
and uniques.date = excluded.date
returning messageid, date, cast(# visitors as bigint) as uniques;
$$;


commit;
```

And here’s an interactive session where we use the newly de ned stored proce-

dure to update our unique visitors hll table. Again, because we are testing mod-

i cations to a data set, we make sure to ROLLBACK our transaction:

```
appdev> begin;

BEGIN

appdev>* select * from tweet.update_unique_visitors();

messageid │

date

│ uniques
```

```
═════════════════╪═══════════════════╪══════════════
3 │ 2018-08-07 │

740

3 │ 2018-08-08 │

258

(2 rows)

appdev>* rollback;

ROLLBACK
```

We can see that it works as we wanted it to, and so we can interactively use this

procedure without having to implement the backend API yet. Our next move

is the following, where we set the daily unique counts for the whole data set we

produced:

```
select * from tweet.update_unique_visitors(100000);
```

The function returns 32 rows, as expected, one per messageid and per day. We

have generated visitors over that period, all on the messageid 3. Note also that

once this command has run, we don’t have any rows in the tweet.visitor table,

as we can check with the following query:

1

```
select count(*)
```

```
from tweet.visitor;
```

This returns zero, of course. In this implementation, the tweet.visitor table is a bu fer of the current activity, and we summarize it in the tweet.uniques table when calling the tweet.update_unique_visitors() function.

Combining Unique Visitors

Now, we can bene t from the nice *hyperlolog set* properties:

```
select to_char(date, 'YYYY/MM') as month,
to_char(date, 'YYYY IW') as week,
round(# hll_union_agg(visitors)) as unique,
sum(# visitors)::bigint as sum
from tweet.uniques
```

group by grouping sets((month), (month, week))

7

order by month **nulls** first, week **nulls** first;

The new function hll_union_agg is an aggregate that knows how to compute

the union of two hyperloglog sets and recognize how many visitors were globally

unique when combining two sets of unique visitors. That’s pretty magical, if you

ask me:

```
month

│

week

│ unique │

sum
══════════╪══════════╪═════════╪════════

2018/08 │ ¤

│

45300 │ 75699

2018/08 │ 2018 32 │

15119 │ 16589

2018/08 │ 2018 33 │
```

```
18967 │ 21461

2018/08 │ 2018 34 │

19226 │ 22104

2018/08 │ 2018 35 │

14046 │ 15545

2018/09 │ ¤

│

18640 │ 21415

2018/09 │ 2018 35 │

6143 │

6299

2018/09 │ 2018 36 │

13510 │ 15116

(8 rows)
```

By using the grouping sets feature here we can make it more obvious how ad-

vanced hyperloglog set support works for unique counting works with the sup-

port of a union operator from multiple sets. In particular, we can see that the

sum of the number of unique visitors would be double-counting a large portion

of the population, which the hyperloglog technique knows how to avoid!

An Interview with Craig Kerstiens

Craig heads up the Cloud team at @citusdata. Citus extends Postgres to be a

horizontally scalable distributed database. If you have a database, especially Post-

gres, that needs to scale beyond a single node (typically at 100GB and up) Craig

is always happy to chat and see if Citus can help.

Previously Craig has spent a number of years @heroku, a platform-as-a-service,

which takes much of the overhead out of IT and lets developers focus on building

features and adding value. The bulk of Craig's time at Heroku was spent running

product and marketing for Heroku Data.

In your opinion, how important are extensions for the PostgreSQL open source

project and ecosystem?

To me the extension APIs and growing ecosystem of extensions are

the bi est advancement to Postgr in probably the last 10 years.

Extensions have allowed Postgr to extend beyond a traditional re-

lational database to much more of a data platform. Whether it's

the initial NoSQL datatyp (if we exclude XML that) in hstore, *to the rich feature set in geospatial with GIS, or approximation al-*

gorithms such

HyperLogLog or TopN you have extensions that

now by themselv take Postgr into a new frontier.

Extensions allow the core to move at a slower pace, which mak

sense. Each new feature in core means it h to be thoroughly tested

and safe. That's not to say that extensions don't, but extensions

that can exist outside core, then become part of the contrib provide

a great on ramp for things to move much faster.

What are your favorite PostgreSQL extensions, and why?

My favorite three extensions are:

1. *pg_stat_statements*
2. *Cit*
3. *HyperLogLog*

pg_stat_statements

easily the most powerful extension for

an application developer without having to understand deep

database internals to get insights to optimize their database.

For many application developers the database *a black box,*

but pg_stat_statements

a great foundation for AI for your

database that I only expect to be improved upon in time.

Citus *: I’m of course biased because I work there, but I followed Ci-*

t

and pg_shard for 3 years prior to joining. Cit

turns Postgr

into a horizontally scalable database. Under the covers it’s sharded,

but application developers don’t have to think or know about that

complexity. With Cit

Postgr

equipped to tackle larger work-

loads than ever before

previously Postgr w

constrained to a

single box or overly complicated architectur .

HyperLogLog *: I have a confession to make. In part I just love*

saying it, but it also mak you seem uber-intelligent when you

read about the algorithm itself. “K minimum value, bit observable *patterns, stochastic averaging, harmonic averaging." I mean who*

doesn't want to use something with all those things in it? In simpler

terms, it's close enough approximate uniqu that are compose-able

with a really small footprint on storage. If you're building something

like a web analytics tool HyperLogLog

an obvio

go to.

How do you typically nd any extension you might need? Well, how do you

know you might need a PostgreSQL extension in the rst place?

[pgxn.org]and github are my two go-tos. Though Google also tends to work pretty well. And of course I stay up to date on new on via

[Postgr Weekly.com.]

Though in reality I often don't always realize I need one. I search

for the problem I'm trying to solve and discover it. I would likely

never search for HyperLogLog, but a search for Postgr approxi-

mate count or approximate distincts would yield it pretty quickly.

Is there any downside you could think of when your application code base now

relies on some PostgreSQL extension to run? I could think of extension's availability in cloud and SaaS o ferings, for instance.

It really depends. There are extensions that are much more bleed-

ing edge, and on that are more mature. Many of the major cloud

providers support a range of extensions, but they won’t support any

extension. If they do support it there isn’t a big downside to lever-

aging it. If they don’t you need to weigh the cost of running and

managing Postgr yourself vs. how much value that particular ex-

tension would provide. As with all things managed vs. not, there

a trade-off there and you need to decide which one

right for you.

Though if something supported and easy to leverage wherever you

run, by all means, go for it.

Part IX

Closing Thoughts

I have written The Art Of PostgreSQL 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*

the world’s most advanced open

source database. I like to say that *PostgreSQL*

YeSQL as a pun, which com-

pares 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 speci cation to testing, including code

review and rewrite.

Of course your journey into *The Art Of PostgreSQL* is only starting. Writing

code is fun. Have fun writing SQL!

Knowledge

of no value unless you put it into practice.

— *Anton Chekhov*

Part X

Index

Index

Document Outline