Introduction to PostgreSQL for the Data Professional

First Edition

By Ryan Booz & Grant Fritchey

Published by Red Gate Books 2024

Title: Introduction to PostgreSQL for the data professional.

Authors: Grant Fritchey and Ryan Booz

Technical Reviewer: Robert Treat

Copy Editor: Louis Davidson

Edition: Preview Edition

Publication Year: 2024

Publisher: Red Gate Books
Cavendish House, Cambridge Business Park
Cambridge, CB4 0XB
United Kingdom

ISBN: 978-1-3999-9678-5

Library of Congress Cataloging-in-Publication Data

Introduction to PostgreSQL for the data professional. / Grant Fritchey and Ryan Booz.
Includes bibliographical references and index.
ISBN 978-1-3999-9678-5

1. PostgreSQL 2. Database Management 3. SQL

Printed in United States of America

About the Authors

Ryan Booz

Ryan is an Advocate at Redgate focusing on PostgreSQL. Ryan has been working as a PostgreSQL advocate, developer, DBA and product manager for more than 22 years, primarily working with time-series data on PostgreSQL and the Microsoft Data Platform. Ryan is a long-time DBA, starting with MySQL and Postgres in the late 90s. He spent more than 15 years working with SQL Server before returning to PostgreSQL full-time in 2018. He's at the top of his game when he's learning something new about the data platform or teaching others about the technology he loves.

Grant Fritchey

Grant Fritchey is a Data Platform MVP and AWS Community Builder with over 30 years' experience in IT, including time spent in support and development. Grant works with multiple data platforms including PostgreSQL and SQL Server, as well as multiple cloud platforms. He has also developed in Python, C#, and Java. Grant writes books for Apress and Redgate. Grant presents at conferences and user groups, large and small, all over the world. He joined Redgate Software as a product advocate in January 2011.

About the Tech Editor

Robert Treat

Robert Treat is a seasoned database professional with nearly three decades of experience working on mission critical, data intensive systems. A passionate open-source advocate and contributor, he is probably best known for his work with Postgres, where he has been recognized as a Major Contributor. His career has traversed a diverse set of organizations including DoorDash, Etsy, Amazon, National Geographic, and WebMD, where he has worked in both leadership and practitioner roles, honing his expertise in databases management and operations. As an independent community advocate, he dedicates his time to fostering open-source initiatives, speaking on a wide variety of topics including open source, devops, and scalable web operations, with a goal of empowering others and driving innovation and operability within the field.

Foreword

It is with great pleasure that I introduce this comprehensive guide to PostgreSQL, authored by two very respected figures in the database community, Grant Fritchey and Ryan Booz. Together, they have produced many blog posts and books about databases for years and years. As you read this book, this is very evident, making it a valuable resource. I have read this book as closely as I have any other book because I served as the copy editor for most of the book.

I have known one of the authors for over 15 years, and the other for 2 now, but I have learned so much about PostgreSQL (and SQL Server and other topics) from them both over these years.

Grant Fritchey, known for his extensive work with SQL Server, has over the past few years, added PostgreSQL to his skill set and brings a wealth of knowledge and practical insights about both to this topic. His experience is not just theoretical; it is grounded in real-world applications and challenges, making his contributions both relevant and actionable. Grant has been a part of the database community for many years now and has always been a wonderful teacher and all-around nice person to deal with.

I met Ryan Booz through working with him at Redgate and he has taught me a lot about how PostgreSQL works (and as a SQL Server expert myself, there are some very interesting differences that aren't always as obvious as you might expect, which is a part of why this book was written!) Everything I have said about how great it is to work with Grant goes exactly the same for Ryan.

Ryan has been instrumental in educating the community through his regular online seminar series, "PostgreSQL 101," and his numerous articles on Simple-Talk.com. His passion for PostgreSQL and his ability to break down complex concepts into understandable terms have made him a trusted voice in the community.

This book is a testament to their hard work and dedication. It provides a very nice introduction to PostgreSQL and includes hints throughout on how it is similar and different from other RDBMS, covering everything from installation and configuration to advanced performance tuning and optimization. Whether you are new to PostgreSQL or looking to deepen your understanding, this book offers valuable insights and practical advice that will help you succeed.

I will go so far as to say that this book was one of the easiest I have ever had the pleasure to edit, and that includes all my own books as well. I am confident that readers will find this book to be an essential addition to their technical library and am proud to admit that here in this foreword.

Louis Davidson (Simple Talk Editor)

Preface

There is no denying that PostgreSQL is growing in popularity. You can look at the history of its growth on the DB-Engines Ranking web site if there's any doubt. As more and more organizations begin to manage some, or all, of their data in PostgreSQL, a growing number of people are going to have to know how PostgreSQL works. We're writing this book for you. Whether you're a Database Administrator (DBA) who has to learn how to maintain a whole new data platform, a developer looking for new and better ways to manage information persistence, or even someone fresh in the IT field looking to expand your skill set, this book is for you. However, that said, we do assume a certain amount of knowledge of databases in general. We have comparisons to other data platforms, frequently Microsoft SQL Server, but others as well, to help establish context. So, this book is more a "beginners in PostgreSQL" rather than a "beginners in databases" in general style book.

Your authors have a very large amount of accumulated knowledge of databases, database management, and database development. We tried to add as much of that into the book as we can. There's guidance for best use of PostgreSQL in a lot of the chapters, not simply descriptions of how things work. We know, based on our own blunders and learning curve, that why you're doing something matters as much as how. So, we put that into the book as well.

Since this is very much an introductory book for PostgreSQL, the best way to get the maximum value from the book is to follow the flow of the book. This is especially true because we use code and structures introduced in earlier chapters, later in the book, so skipping around could cause confusion. However, most chapters stand on their own, for the most part. While it is certainly possible to skip around, you may hit snags when you run the sample code. Speaking of sample code, you'll generally see it looking like this:

Listing X-1. This is sample code

```
SELECT cola FROM sometable;
```

The code will be introduced in some manner, you'll see it in a different format, and you'll see the Listing caption with a chapter and the number of the listing within the chapter. This makes it possible to refer back to code within a chapter. You will also see code that's shown in-line with the text, just like `sometable` here. You'll note that this has a different font, to help set it apart from the rest of the text. Speaking of Notes, and Warnings, you'll see this throughout the book:

NOTE/WARNING/CAUTION: These are points that we don't want to get lost in the text because they're important. This way, you can spot them easily.

We broke the chapters down into sections that will have headings to let you know what they're about. Hopefully this helps navigation as you read the book.

We are both excited to share this book with you. There are several reasons for this. While writing a book is somewhat difficult, and very time consuming, when you're done, there's a real sense of accomplishment. Also, both of us have run into issues while learning PostgreSQL and thought to ourselves, if only someone had pointed this out for me. Well, we've tried to do that throughout the book in order to help you on your journey. Finally, and most important, we both really enjoy being able to help others. We enjoy it even more when it's helping others get going on the PostgreSQL data platform, because we've had a lot of fun exploring this space. Our goal is to help you on that journey too. Thanks for reading.

2
PostgreSQL Basics and Differences

Now that you have an overview of how PostgreSQL began and the unique process for developing and maintaining features, it's time to start making a few comparisons between the knowledge you already have, and how it translates to PostgreSQL.

Fortunately, moving from one relational database system to another already puts you at an advantage because you know a dialect of SQL, understand the basics of schema design, and principles like backups and high availability. However, without a little help upfront, it can be challenging to figure out what the differences are which may unexpectedly trip you up as you begin your journey to learn PostgreSQL. This chapter aims to help uncover, at a high level, some of the major features and syntax differences to give you a head start.

Feature Comparison

Trying to highlight all the differences between relational databases could easily fill an entire book. Because we both come from SQL Server backgrounds, we'll be specifically comparing features in SQL Server to narrow the focus. We want to highlight features you may currently be using that work differently (or don't exist at all), and some features that PostgreSQL brings to the table that a database platform like SQL Server doesn't have.

Extensibility

While we'd prefer to start this chapter talking about features in SQL Server and their counterparts (or not) in PostgreSQL, talking about the extension capabilities in PostgreSQL are essential for some of the comparisons that follow.

At the end of Chapter 1 we briefly discussed two of the reasons why PostgreSQL has enjoyed so much popularity and fervor over the last decade. Aside from licensing, the ability to use extensions to add functionality or modify how a process within Postgres behaves has proven to be extraordinarily powerful for a wide array of use cases.

To a user of SQL Server, this kind of flexibility is unheard of. As a licensed product from Microsoft, users are dependent on their timeline and feature goals. If adding new AI features

will be helpful and potentially profitable, it's likely that the next release will have a significant amount of effort dedicated to making AI accessible within the database. However, if it was important to you that SQL Server supported more native datatypes for geospatial data, including special indexes that make searching and correlating that data easier, you may be out of luck. That's not a specific critique of SQL Server or the development team (you're doing great!). It's often just an economic decision.

With PostgreSQL, however, that geospatial workload is easily employed by installing the PostGIS extension. This extension creates new datatypes and bindings to special indexes that work better with geospatial data. It also creates dozens of functions that are unique to geospatial data and allow a myriad of features to find the distance between two points, convert geospatial datatypes, do nearest neighbor search of points, and more.

The same goes with features like indexes. PostgreSQL has a set of hooks that any extension developer could take advantage of to create new index types. Other extensions might create a listener so that when a query plan is being created, the extension has an opportunity to modify something about the plan because of the kind of data the query is accessing.

Extensions can be created in any supported language on the server (including SQL) and can be used for special or mundane tasks. Many PostgreSQL administrators create extensions that simply install a known set of administrative functions that their teams use across the estate. Because extensions are versioned, it's an easy and safe way to determine if a particular database has the latest version of maintenance functions specific to that company or team.

The growing extension landscape within the PostgreSQL community is a powerful way to add functionality that wouldn't otherwise be possible, all without waiting on the (often) multi-year cycle for adding new, major features to the core PostgreSQL engine. While that may sound scary, the most popular extensions have been very well vetted for functionality, stability, and security best practices.

Talking more deeply about extensions, how to install them, manage them, and use them effectively could easily take a few dozen pages of a book all by itself. However, all major cloud providers support the most popular extensions and provide excellent documentation to help get you started. As of this writing, there are over 1,000 known extensions in the wild, but only about 50-75 that are heavily used to help solve common problems.

There are multiple places to find extensions, however, as of 2024, there is still no one central place to find, download, and install all of them. A few places to start include:

- PGXN, the Postgres Extension Network (pgxn.org)
- Postgres Trunk (pgt.dev)
- Postgres Extension Repos List (https://github.com/joelonsql/postgresql-extension-repos)
- Linux package managers (rpms, Debian, homebrew, etc.)

The main takeaway is that PostgreSQL supports extensions which can provide new functionality or enhance existing features of PostgreSQL. No other relational database supports this kind of feature and is a primary, distinguishing factor of PostgreSQL. You'll hear about a few extensions in the next few sections.

Built-in Job Scheduler

SQL Agent jobs are a mainstay of how so much background work is accomplished in the management of SQL Server and tasks that keep your data workflows in good working order. Whether you run Ola Hallegren's maintenance scripts or schedule some type of ETL jobs, having a built-in scheduler as part of the server is convenient and often essential. With SQL Agent jobs you also benefit from features like notifications or server email integration. In our experience, agent jobs are used on nearly every production SQL Server.

PostgreSQL doesn't have an internal, out of the box, job scheduling mechanism. Instead, this functionality can be provided by numerous extensions.

The most popular, and supported by all major cloud providers, is pg_cron. There is no graphical interface for setting up the scheduled jobs, and that currently holds true for any of the popular extensions that provide this functionality. But, with minimal effort, the extension can be installed and jobs created with very specific cron-like scheduling parameters.

Query Hints

Love them or hate them, query hints are a necessity for many of the things we do in SQL Server. Whether MAXDOP, OPTIMIZE FOR, RECOMPILE, or NOLOCK (sorry, we had to do it), there are good and valid reasons for many of the hints when used appropriately.

PostgreSQL does not come with native support for query hints. You cannot force a specific join type, degree of parallelism, or point to a specific index when writing PostgreSQL queries. Unless, of course, you install the `pg_hint_plan` extension, also available on most major cloud providers.

With `pg_hint_plan` installed in the database you can add Oracle-style query hints to your queries. Hints are added to the SQL query as comments with specific commands that can direct the planner to join tables in specific ways, use an index, modify row estimates for a relation, and more.

Query Plan Cache

In SQL Server, the query plan cache is an essential feature that helps reduce compilation overhead for queries. Rather than compiling a plan on every execution, the query planner can pull a matching query plan out of the cache in some circumstances and get on with execution. There are pitfalls with the query plan cache that still exist today like parameter sniffing of stored procedures, but any time work can be reduced on the query planning and execution process is a win.

PostgreSQL does not have a query plan cache. Every execution of a query produces a new plan, unless you specifically prepare a query statement and reuse that plan for future queries. Admittedly, however, how and why you would use prepared statements is a slightly complex topic and one you should study before implementing in specific workloads you might have.

The idea that every query requires compilation is a foreign concept to SQL Server developers and DBAs. We're taught over and over that the plan cache is essential to reduce overhead for the server to reduce every possible CPU cycle of work. Interestingly, query planning is rarely the issue that slows down a query or overloads the server in PostgreSQL.

We don't currently know of any plans to support a query cache in PostgreSQL core, nor any popular extensions that do. However, some cloud vendors have implemented their own features that do something similar (i.e. AWS Aurora PostgreSQL query plan management), and there are a few publicly available extensions that attempt to implement similar functionality, although none of them are regularly discussed or unavailable in hosted environments.

Execution Plan Viewer

This falls more under tooling than a deficiency in viewing query plans, but having graphical viewers that provide various metric breakdowns is a staple for SQL Server developers. The ease of opening any plan in a detailed viewer is front and center in SQL Server Managements Studios (SSMS) and even Azure Data Studio. Because these tools are included as part of the products supported and maintained by the team developing SQL Server, it's an integrated experience that we often take for granted.

Depending on the IDE that you use with PostgreSQL, a graphical query plan viewer may be available. PGAdmin has included one for many years, and there are multiple online tools that will turn the text-based plans into a visual plan, but it's just not as seamless as what you know in SSMS or the handful of other go-to tools.

Regardless of the tool or method you might use to produce a visual plan, you'll still be better served in the long run to become familiar with text based EXPLAIN plans. It's not as scary as it sounds once you know a few basics. We'll explore some of the basic concepts and a few online tools to help parse query plans in Chapter 11: Indexes and Query Tuning.

Terminology Differences

Although PostgreSQL and SQL Server are both relational databases that support many features of the ANSI SQL standard, knowing the correct terminology for similar concepts is essential, particularly as you search documentation or ask for help from the community. In the next section, we'll look specifically at differences in SQL concepts.

Cluster vs. Instance

Like the choice to rename the project to PostgreSQL, forever confusing users about how to pronounce it, the term used to identify the running PostgreSQL process is slightly confusing, too.

Technically, as referenced in the official documentation, the running process is called a *cluster*, not *instance*. You can have multiple running clusters on a host, each listening to a different port, and configured differently. Obviously, this can cause some confusion when all distributed database systems are a cluster of, well, clusters. We hear more people call an

individual PostgreSQL process an instance as time goes by, but it's worth noting that when you typically see the word "cluster" in reference to a PostgreSQL server, that's just a single running process unlikely to be connected to other instances.

Role vs User

In PostgreSQL, all principles that can connect to a database are called *roles*. There is no technical difference between a role that functions as a user, and a role that functions as a group. By convention, all user roles are allowed to login, while group roles are not. However, there's nothing that enforces that in PostgreSQL. Depending on how a role is configured, it may be able to grant membership to other roles, so even if the role can login (i.e. a "user" role), it might still offer membership to a group of other roles.

One other thing of note is that although roles are created at the cluster level, a user role can only establish a connection to a database. Even if you supply the correct credentials, if the role doesn't have at least one database that it has the privilege to connect to, they will not be able to establish a connection.

Chapter 6 is dedicated to roles and security.

Tuple vs. Row

In SQL Server, a row, is a row, is a row… for the most part. In PostgreSQL, however, you'll often hear the word *tuple* to describe the individual rows of data returned from a table. They essentially mean the same thing, but there's a (historically) academic reason why the word tuple is still used.

A tuple refers to the schema (column types, widths, order, etc.) of a row. Rows are instances of tuples. But using the word tuple means that you're not talking about just an abstract set of data being returned. It must match and adhere to the constraints of the schema of the table, which is the definition of a tuple.

It's a very minor difference, and honestly doesn't matter in the grand scheme of things, but you'll hear the word tuple a lot, so it's worth understanding that it's just a row, which is a representation of the schema from the table.

Literal Object Qualifiers

The ANSI SQL standard says that double quotes should be used to qualify objects (generally), but especially when objects don't adhere to the naming rules, like adding spaces in an object name. Unfortunately, not all databases adhere to this standard and choose other qualifying characters.

By default, SQL Server uses square brackets to qualify names (left) and PostgreSQL uses double quotes (right).

```
[Table One] vs. "Table One"
```

This is especially noteworthy when migrating to PostgreSQL with lots of legacy SQL that uses a different qualifier.

COPY vs. BULK INSERT

Loading lots of data from external sources is necessary for almost every database project in modern applications. Dating back to the original, open-source research project that is the foundation for PostgreSQL, there has been a special SQL command called COPY that only PostgreSQL supports.

COPY is generally supported by any database which is derived from PostgreSQL and is the preferred method for inserting lots of data quickly. The binary COPY protocol is also supported by most modern language SDKs to assist with the speedy loading of data.

TOAST

The term TOAST stands for "The Oversized Attribute Storage Technique". (Clever, eh?)

TOASTing data is essentially off row storage for data that cannot fit into a single 8Kb page. Any variable length data type can be toasted, which writes the data to other pages on disk and then creates a pointer in place of the data within the row. Any time a query requests that specific data, it is accessed outside of the row, materialized, and returned.

TOASTed data is automatically compressed, with recent versions of PostgreSQL supporting user-selectable compression algorithms if they are made available during the compilation of the PostgreSQL kernel.

While that doesn't sound like anything fancy (all database servers need a way to store large data outside of a data page), the unique feature in PostgreSQL is that there are access methods which extensions can hook into when data is toasted. With this hook, extensions like TimescaleDB can implement a proprietary data compression feature which stores data off in an aggregated form which can help time-series type workloads.

Just don't get out the butter and jam when someone using PostgreSQL starts talking about TOAST.

SQL Differences

Finally, we want to spend just a few pages discussing some of the first SQL differences that are likely to trip you up if you're coming from SQL Server.

Data Types

PostgreSQL has many built-in data types and more can be added through extensions or through direct SQL statements. New index types can also be created to assist the query planner in retrieving data more efficiently. However, the standard set has a few surprises that we wanted to highlight.

TEXT

PostgreSQL does support the CHAR/VARCHAR datatypes, but over the years there has been a lot of testing and discussion around the need for anything other than TEXT. As

counter intuitive as it may sound, there's almost never a reason to use a character datatype instead of TEXT. It's UTF-8 by default, automatically takes care of any toasting and off-row compression, and it simplifies schema development.

One of the main reasons SQL Server developers have been told time and again ***not*** to use VARCHAR(MAX), which would essentially be equivalent of using TEXT, is because the query planner can do unexpected things with memory grants if all text fields are set as "big" fields.

PostgreSQL doesn't deal with query memory in the same way, so this isn't a concern. Also, disk storage and data retrieval are not negatively impacted when using text fields. If you require a specific maximum width for a text column, add the appropriate constraint to prevent wider data from being saved to the text field. Using a constraint also has the added benefit of being able to widen or remove the constraint at any time without having to modify the datatype and potentially cause a table re-write.

TIMESTAMP (WITH TIME ZONE)

TIMESTAMP (`timestamp`) or TIMESTAMP WITH TIME ZONE (`timestamptz`) are generally equivalent to the DATETIME/DATETIMEOFFSET datatypes in SQL Server. Like SQL Server, time zone aware datatypes will store the data in UTC.

The output of a TIMESTAMP WITH TIME ZONE will be converted to the time zone set within the session. Depending on the tool/IDE you use to query `timestamptz` data, your time zone may be read from your computer locale or from a configurable setting.

Using `psql`, the queries in Listing 2-1, 2-2, and 2-3, demonstrate how the time zone of the session modifies the output of a query.

Listing 2-1: *Show the current time zone of the session*

```
SHOW timezone;

TimeZone|
--------+
Etc/UTC |
```

Listing 2-2: *Timestamp with time zone value stored as UTC*

```
SELECT last_update FROM bluebox.customer
ORDER BY customer_id LIMIT 1;

last_update                  |
-----------------------------+
2023-10-13 20:41:34.320464+00|
```

Listing 2-3: *psql displays output of the timestamptz column at session time zone*

```
SET TIME ZONE 'EST';

SELECT last_update FROM bluebox.customer
ORDER BY customer_id LIMIT 1;

last_update                  |
-----------------------------+
2023-10-13 15:41:34.320464-05|
```

The problem with the time zone used for the output is that different tools handle it different ways. For example, we often use DBeaver to execute queries in PostgreSQL. Unfortunately, it sets the time zone at startup based on the locale of your computer. Attempting to set the time zone of the session as shown above does not affect the output formatting without additional configuration. Therefore, someone else running the same query in a different time zone will get a different output with a different offset value. The timestamp value itself has not changed; it just looks different since each user will "see" the data in their locally specified time zone.

If you need to guarantee that the output is formatted using a specific time zone offset, then you must cast the value within the query. The SELECT statement in Listing 2-4 will produce the same timestamp value and offset for anyone that runs it, regardless of their session time zone.

Listing 2-4: *Cast a timestamptz to a specific time zone*

```
SELECT last_update AT TIME ZONE 'EST' FROM bluebox.customer
ORDER BY customer_id LIMIT 1;

last_update                  |
-----------------------------+
2023-10-13 15:41:34.320464   |
```

Storing and querying dates and timestamps can be complicated at times. However, for consistency and ease, using TIMESTAMP WITH TIME ZONE (`timestamptz`) to store

timestamp values is the most reliable. It's flexible and will always output the appropriate value given a specific time zone.

ARRAY

One of the most controversial datatypes, specifically from users coming to PostgreSQL from a different database, is the array datatype. Any datatype in PostgreSQL can be represented in array form as a column type. It certainly takes some getting used to, and it's not a datatype to be used without purpose, but it can be powerful when used correctly.

Arrays should (generally) never be used as a replacement for a linking table, but when a list of items can be associated with an object, this has a few advantages.

First, arrays are type aware. Rather than using a text field to store a comma-separated list of values that might store text, integers, floating points, timestamps, and more, arrays know the type and will error if a user tries to insert integers into a text field.

Arrays can also be indexed in a way that goes beyond a comma-separated text field and there are specific built-in functions to query and manipulate arrays. Also, as helpful as array types can be when used correctly to store data in a table column, it's often a game changer to be able to use array datatypes in pl/pgsql (analogous to TSQL for SQL Server) for development.

Case Rules

PostgreSQL uses case sensitive collations which means that all object names are case sensitive. However, PostgreSQL automatically transforms all object names to lowercase unless they are qualified with double quotes. If you don't know about this ahead of time, especially coming from SQL Server, a lot of queries will start to fail in unexpected ways.

For instance, let's say that you create a table using the code in Listing 2-5.

Listing 2-5. *Creating a table object without qualifying the name*

```
CREATE TABLE CamelCase (c1 text);
```

Because the table name is not qualified with double quotes, any of the select statements in Listing 2-6 will return data because the name is stored in lowercase, and all object references in these queries transform the table name to lowercase.

Listing 2-6. *Unqualified object names will query as lower case*

```
SELECT * FROM CamelCase;
SELECT * FROM camelcase;
SELECT * FROM CAMELCase;
SELECT * FROM CaMeLcAsE;
```

However, the SELECT statement in Listing 2-7 will fail because the qualified table name instructs PostgreSQL to find that exact table, with the exact name and case.

Listing 2-7. *The qualified object name doesn't exist because it was created unqualified*

```
SELECT * FROM "CamelCase";

ERROR:  relation "CamelCase" does not exist
LINE 1: select * from "CamelCase";
```

This is especially troublesome when two different developers execute SQL to generate tables but one qualifies the name and the other doesn't. Listing 2-8 shows how two different tables will be created with no errors.

Listing 2-8. *Mixing qualified and unqualified object names*

```
-- This results in a table with the matching case
CREATE TABLE "CamelCase" (c1 text);

-- This results in a table with the object name of "camelcase"
CREATE TABLE CamelCase (c1 text);
```

Instead of using camelCase for object names, the majority of PostgreSQL functions and schemas utilize lower snake case to name objects, using underscores between words rather than uppercase letters. This means that the preferred method of creating the example table above would be to use the naming shown in Listing 2-9.

Listing 2-9. *Most PostgreSQL developers prefer snake case object names*

```
CREATE TABLE camel_case (c1 text);
```

Between object name qualifiers and the automatic transformation to lowercase, we've seen many queries rewritten from TSQL that produce errors simply because of this principle.

Don't use camel case in PostgreSQL. You will have to qualify the name on creation of the object and every query that ever references that table, column, function, and more.

LIMIT vs. TOP

To select a limited number of rows in a query, SQL Server users instinctively use type `SELECT TOP(10) FROM…` and it's a hard habit to break. Instead, PostgreSQL uses the more standard LIMIT keyword to limit the number of rows returned from a query. Listing 2-10 and 2-11 demonstrate how to select 10 rows of data from the rental table from each database.

Listing 2-10. *Retrieve 10 rows from the rental table in SQL Server*

```
SELECT TOP(10) * FROM [Bluebox].[Rental] ORDER BY RentalID;
```

Listing 2-11. *Retrieve 10 rows from the rental table in PostgreSQL*

```
SELECT * FROM bluebox.rental ORDER BY rental_id LIMIT 10;
```

DATE_PART vs DAY/MONTH/YEAR

Like the TOP function in SQL Server, we're guessing that most of you have some muscle memory around how you get the parts of a date out of a timestamp-like value. The sensible functions DAY(), MONTH(), and YEAR() are clear and commonplace in everyday TSQL. PostgreSQL on the other hand uses the date_part() function to return various parts of a timestamp or date value. (SQL Server has a datepart() function as well).

Slightly more cumbersome at first glance, but this function also accommodates a few dozen parameter values to extract every part of the source value: `day`, `month`, `year`, `century`, `doy`, `epoch`, and others that allow you to get just the data you need.

For example, a query that counts the number of DVD rentals per year is shown in Listing 2-12 for SQL Server and Listing 2-13 for PostgreSQL.

Listing 2-12. *Extracting the year from a timestamp in SQL Server*

```
SELECT COUNT(*), YEAR(RentalStart) RentalYear
FROM BlueBox.Rental
GROUP BY RentalYear
ORDER BY YEAR(RentalStart) DESC;
```

Listing 2-13. *Extracting the year from a timestamp in PostgreSQL*

```
SELECT COUNT(*), DATE_PART('year',rental_start) rental_year
FROM bluebox.rental
GROUP BY rental_year
ORDER BY rental_year DESC;
```

Note that in PostgreSQL, you can use the function alias in both the GROUP BY and ORDER BY clauses. In SQL Server, aliases are only allowed in the ORDER BY clause.

INTERVAL vs DATEADD

PostgreSQL has an INTERVAL datatype which represents an interval of time. And although there are functions like `date_add()` and `date_subtract()`, PostgreSQL Date/Time values support direct math using an INTERVAL.

To query for the last week of data from the rental table using dynamic date math, Listings 2-14 and 2-15 show how this can be accomplished in SQL Server and PostgreSQL.

Listing 2-14. *Dynamic date math in SQL Server using DATEADD*

```
SELECT COUNT(*), YEAR(RentalStart)
FROM BlueBox.Rental
WHERE RentalStart > DATEADD(MONTH,-1,SYSDATETIME())
GROUP BY YEAR(RentalStart)
```

Listing 2-15. *Dynamic date math in PostgreSQL using the INTERVAL type*

```
SELECT COUNT(*), DATE_PART('year',rental_start) rental_year
FROM bluebox.rental
WHERE rental_start > NOW() - INTERVAL '1 month'
GROUP BY rental_year;
```

LATERAL vs APPLY

As you grow in skill developing against SQL Server, eventually you learn about the APPLY operator in TSQL joins. It's an effective (and often efficient) way to iterate over each row in an outer query against a secondary query that references data from the outer row. If you ever want the most recent value for each item in a query, APPLY is usually the easiest way to do that, no extra function required.

PostgreSQL subscribes to the ANSI SQL standard approach to this called a LATERAL query. While there are a few minor differences in how LATERAL queries can be written and used, the overall concept is the same.

Listing 2-16 and 2-17 show an example query to retrieve the customer ID from each store with the most recent rental for SQL Server and PostgreSQL respectively.

Listing 2-16. *SQL Server In-line code*

```
SELECT StoreID, CustomerID, RentalStart
FROM BlueBox.Store s1
CROSS APPLY (
  SELECT TOP(1) CustomerID, RentalStart
  FROM  BlueBox.Rental
  WHERE StoreID = s1.StoreID
  ORDER BY RentalStart DESC
) r1;
```

Listing 2-17. *SQL Server In-line code*

```
SELECT store_id, customer_id, rental_start
FROM bluebox.store s1
INNER JOIN LATERAL (
  SELECT customer_id, lower(rental_period) AS rental_start
  FROM bluebox.rental
  WHERE store_id = s1.store_id
  ORDER BY rental_start DESC
  LIMIT 1
) r1 ON true;
```

Anonymous Code Blocks

Last but not least, we need to talk about writing ad hoc, inline code with variables, functions, loops, and more. SQL Server developers and DBAs are used to doing this everywhere. Whether in SSMS or a TSQL script that will be run as part of a deployment, we create code blocks liberally, because it's just so useful to efficiently doing complex work.

The challenge is that this kind of procedural programming within a SQL script is not supported by the ANSI SQL standard. Any time you write scripts using variables and more for SQL Server, they are being executed by the TSQL engine which understands that some of the code is treated differently to provide added value. This works because SQL Server defaults to executing SQL files with the TSQL engine unless indicated otherwise. We just never have to think about it.

PostgreSQL treats SQL as the declarative language that it is, without inherent ability to execute procedural code. Instead, PostgreSQL expects most procedural code to be written in a function which declares the procedural language to use. Using variables and other procedural code directly in the script simply won't work.

However, you can declare what's known as an anonymous code block (sometimes called "DO" blocks) directly in your SQL script to accomplish some of the same functionality. This is meant for one-off situations and imposes some annoying limitations, like not allowing the block to return a set of data.

As a very simple example, Listing 2-18 and 2-19 demonstrates how to declare a variable, select a value into it, and then print the results.

Listing 2-18. *SQL Server In-line code*

```
DECLARE @LimitedDescription AS NVARCHAR(50);
SELECT @LimitedDescription = LEFT(Overview,50)
  FROM BlueBox.Film
  WHERE FilmID = 100;

PRINT @LimitedDescription;
```

Listing 2-19. *PostgreSQL Anonymous Code Block*

```
DO $$
   DECLARE limited_description TEXT;
BEGIN
  SELECT LEFT(overview, 50) FROM bluebox.film
  WHERE film_id = 100
  INTO limited_description;

  RAISE NOTICE '%', limited_description;
END;
$$
```

Certainly not as straightforward as you might be used to, and as we said, there are other limitations that will probably encourage you to just write a function. But when you need to do some ad hoc data processing that doesn't require the return of a dataset, anonymous code blocks are very helpful.

Conclusion

In this chapter we tried to highlight some of the main differences between PostgreSQL and SQL Server. As we mentioned at the beginning of the chapter, your current skills writing SQL and using a relational database give you nearly all the tools you need to effectively use PostgreSQL. Often just knowing what to call something can make all the difference when you are starting out or need to ask for help.

There are plenty of other things that differ between the two database platforms, but we wanted to keep this chapter focused and give you some new tools to start exploring with.