Chapter 13
Performance Tuning

Introduction

Any **Relational Database Management System** (**RDBMS**) intends to resolve user queries as quickly as possible, and PostgreSQL is no exception. So, keeping postgres well performant is one of the main goals for **Database Administrators** (**DBAs**).

In previous chapters, we have studied many of the capabilities of PostgreSQL and how to configure and use them. Now, we will dive into the performance tuning topics. We will learn about the basic concepts and components involved in database tuning and some best practices we can use as an initial guide.

Structure

During this chapter, we will study the next sections:

- Indexes
- Statistics
- Explain plan
- Best practices for the `postgresql.conf` parameters

Objectives

Once you complete this chapter, you will be familiar with the basic components for tuning in PostgreSQL and understand their relationship and impact. Also, you will relate the standard best practices when installing or tuning a postgres system to deliver the best performance.

Indexes

Have you ever noticed the book indexes that help locate the chapters by mentioning the page number of the chapter? One can jump to that page directly rather than going through each chapter sequentially which takes a lot of time. In the same way, indexes also create a similar kind of metadata like page numbers which helps to jump to the requested data directly instead of scanning the whole table sequentially.

This benefit comes with an overhead since adding an index involves it being created / modified as data gets inserted, updated, or deleted from the table. This is one of the reasons one should create wisely through proper analysis. If the table is used for read-only operations more than the write operations, then it could be beneficial to create the indexes; else, it might become an overhead to update the index on every write operation.

Syntax as per PostgreSQL Documentation: *(Reference - PostgreSQL Community Index)*

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
[ ONLY ] table_name [ USING method ]

    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass
    [ ( opclass_parameter = value [, ... ] ) ] ] [ ASC | DESC ] [ NULLS
    { FIRST | LAST } ] [, ...] )

    [ INCLUDE ( column_name [, ...] ) ]

    [ NULLS [ NOT ] DISTINCT ]

    [ WITH ( storage_parameter [= value] [, ... ] ) ]

    [ TABLESPACE tablespace_name ]

    [ WHERE predicate ]
```

Generic Syntax normally used:

```
CREATE INDEX <schema_name>.<index_name> on <table_name> (<column/
column-list>)
```

CREATEing and DROPping the index is not an online activity by default. The table is locked for any modification. However, read operations can be performed while the index is being created/modified by including the **`CONCURRENTLY`** keyword in the index creation statement. It will help to modify indexes concurrently online, as shown in the syntax as per PostgreSQL documentation above.

Indexes on multiple columns can also be created, which are called Multi Column Indexes. While creating such indexes, make sure that the order of columns is the same as used in the query's **`WHERE`** clause.

Reindex

Reindexing an index is one of the maintenance activities in PostgreSQL, which helps to rebuild the corrupted/bloated/invalid index concurrently. An index can also be rebuilt when storage is changed for the specific index.

Syntax as per PostgreSQL Documentation: *(Reference - PostgreSQL Community Re Index):*

```
REINDEX [ ( option [, ...] ) ] { INDEX | TABLE | SCHEMA | DATABASE |
SYSTEM } [ CONCURRENTLY ] name


where option can be one of:

    CONCURRENTLY [ boolean ] – Reindex can be created concurrently
    online

    TABLESPACE new_tablespace – Reindexing could be done in  new
    tablespace

    VERBOSE [ boolean ] – Logs will be printed while reindexing
```

Whenever a Primary Key or Unique Key constraints are created, by default the system creates indexes on such columns as commonly those columns will be extensively used for fetching the data in the queries. Generally, the indexes could be created on columns used to filter data, that is, `WHERE` clause columns. It is not a thump rule; however, it is a good practice to create indexes on those columns that are used as the child table's foreign key. Typically such columns will be used in join conditions to fetch the data.

Indexes might not be needed for small databases initially, as default indexes created on Primary Key or Unique are enough. However, as time goes on, data also increases, and it takes more time than it usually takes. One can use **`EXPLAIN PLAN`** to find out

whether indexes are used by the query to fetch the data. **EXPLAIN PLAN** is discussed in detail later in this chapter.

Indexes are one of the building blocks to improve the performance of the queries. These queries could be the reporting queries that fetch data or any **SELECT** queries that need improvement in the query execution time.

Indexes might also make the performance of **INSERT, UPDATE**, and **DELETE** queries in which a **WHERE** clause is used for search conditions. The **ANALYZE** command is recommended so that statistics of the indexes can be kept up to date, and whenever it is used, it will contain updated metadata which will help to enhance performance better.

Index types

Each query is different and may not give the best results with the default BTREE index. To overcome this, PostgreSQL has different types of indexes, which help increase the performance of various kinds of queries. If one wants to create a specific index type apart from the default BTREE index, it can be done with the **USING** keyword in the index creation statement. Let us now understand the different types of indexes in PostgreSQL.

Btree index

As discussed earlier, by default BTREE index is created when no index type is mentioned in the **CREATE INDEX** statement. The b-tree index sorts the data and keeps it sequentially. It works well when the *equal-to* operator is used in the comparison in the query's **WHERE** clause. Along with *equal-to* BTREE works well with the below operators as well:

- <
- >
- <=
- >=
- IS NULL
- IS NOT NULL
- BETWEEN
- IN

B-tree indexes can also be used to retrieve data in sorted order. This is not always faster than a simple scan and sort, but it is often helpful.

Hash index

Hash Index generates a 32-bit hash code on the column where the index is created. It works best when the *equal* operator is used in the where clause of the query.

GiST and SP-GiST index

These indexes are used when two-dimensional geometrical data types are used.

Gin index

This index is used when one-dimensional data types like ARRAYS are used in the data types.

Brin index

BRIN indexes (a shorthand for **Block Range INdexes**) store summaries about the values stored in consecutive physical block ranges of a table. Thus, they are most effective for columns whose values are well-correlated with the physical order of the table rows. *(Reference: PostgreSQL Community Index Types).*

Indexes and expressions

An index column need not be just a column of the underlying table but can be a function or scalar expression computed from one or more columns of the table.

For example,

```
—-WHERE clause with expression

SELECT * FROM people WHERE (first_name || ‘ ‘ || last_name) = ‘John
Smith’;


—-INDEX creation with expression same as WHERE clause

CREATE INDEX people_names ON people ((first_name || ‘ ‘ || last_name));
```

The system sees the query as just *WHERE indexed-column = ‘constant’* and so the speed of the search is equivalent to any other simple index query. Thus, indexes on expressions are useful when retrieval speed is more important than insertion and update speed.

The metadata views - **pg_index** and **pg_indexes** give information about the details of the indexes like the schema name, index name, and the like.

We will see more examples of INDEX later in this chapter.

Statistics

Statistics are what we usually call metadata, meaning *the data about the data*. As we saw in the *Query Processing* section of *Chapter 6*, the planner is the stage in charge of determining the fastest and cheapest path to get the data required by a query. To accomplish that, it relies on the collected statistics.

There are two main types of statistics in PostgreSQL, each maintained by different components.

- The statistics about the server activity. These are collected and maintained by the stats collector background process; it feeds a set of catalog views with information about the usage of the system objects, such as the count of accesses to the tables and indexes or the number and state of the user sessions. Usually, this information is used by the Database Administrator to verify the system's state.
- The statistics about the data distribution within the tables and indexes. This information is collected when the **ANALYZE,** or **VACUUM ANALYZE** commands are executed manually or by the autovacuum background process. This is the information the planner uses, and we will focus on this section.

The statistical information about the tables and indexes data is stored in three different system catalogs, depending on their kind:

- **pg_class**: In this catalog, the system stores statistics about the number of tuples (rows) in the tables or indexes and the occupied blocks in the disk, alongside some objects' identity details.
- **pg_statistics**: Here postgres stores information about the *selectivity* of the data. It has one or two rows per table column, depending if the table has child tables (inheritance). This information might be hard to read since it is intended to be used by the system itself; however, postgres has a view called **pg_stats** which shows the same information in a more easy-to-understand way.
- **pg_statistics_ext_data**: Here additional information is stored in the case we add an *extended statistics object* for a specific set of tables columns.

In the following paragraphs, we will study each of these in detail and see a few examples.

To illustrate the different statistics types, we will use an example database called **pagila**, a port from the MySQL **sakila** example database. *(Reference: Devrim Gündüz pagila).*

Statistics in **pg_class**

Considering the tables and data from the example database **pagila** we can review the statistics in the **pg_class** catalog. For example, consulting the information for the address table and its index:

```
pagila=# SELECT
  relname AS “relation_name”,
  relkind AS “relation_kind”,
  reltuples AS “relation_tuples”,
  relpages AS “relation_pages”,
  pg_size_pretty(pg_relation_size(oid)) AS “relation_size”
FROM pg_class
WHERE relname LIKE ‘address%’
AND relkind IN (‘r’, ‘i’);

-[ RECORD 1 ]---+-------------
relation_name   | address
relation_kind   | r
relation_tuples | 603
relation_pages  | 8
relation_size   | 64 kB
-[ RECORD 2 ]---+-------------
relation_name   | address_pkey
relation_kind   | i
```

```
relation_tuples | 603
relation_pages  | 4
relation_size   | 32 kB
```

This catalog shows that the table and the index have 603 rows. The table uses 8 pages, each of 8 KB occupying 64 KB in total in the disk. And, how we could guess the index is smaller and uses only 4 pages with a total of 32 KB. The planner uses this information to know how much work it would take to read from these objects.

The number of tuples is not updated in real time. As we saw before, the statistics are updated by the **ANALYZE** or **VACUUM ANALYZE** commands, but the planner uses the information about the used pages to escalate the estimation of the total number of rows, so the approximation is closer to the actual number.

Statistics in pg_statistics

The information in the **pg_class** is OK when the queries aim to read the complete table data, so the planner can know the total number of rows it will retrieve. However, the most common operations are executed to retrieve just a subset of rows from the table.

To improve these filtering operations, using the **WHERE** clause, postgres keeps statistics about the selectivity of the values contained per table column. This information is stored in the **pg_statistics** catalog, and as we saw above, there is a more human-readable view called **pg_stats**. Considering the same sample database, we could verify the information of the **postal_code** column from the **address** table:

```
pagila=# SELECT
  attname AS “column_name”,
  n_distinct AS “distinct_rate”,
  array_to_string(most_common_vals, E’\n’) AS “most_common_values”,
  array_to_string(most_common_freqs, E’\n’) AS “most_common_
  frequencies”
FROM pg_stats
WHERE tablename = ‘address’
AND attname = ‘postal_code’;
```

```
-[ RECORD 1 ]-----------+-------------
column_name             | postal_code
distinct_rate           | -0.9900498
most_common_values      |             +
                        | 22474       +
                        | 52137       +
                        | 9668
most_common_frequencies | 0.006633499 +
                        | 0.0033167496+
                        | 0.0033167496+
                        | 0.0033167496
```

The **pg_stats** view has some other details, but in this example, we can see the rate of distinct values, the most common values, and the frequencies of the most common values in the **postal_code** column. *(Reference: PostgreSQL Community pg_stats)*

Analyzing more details on this information:

- The **distinct_rate** shows the proportion of distinct values versus the total rows. In the case of a unique value column, such as the Primary Key, this rate is 100%, and it is represented with the value -1. The table column from the example is near -1, meaning almost all the values are distinct.
- The **most_common_values** gives insight into the **postal_code** values that repeat the most. In this case, four values: **null, 22474, 9668**, and **52137**.
- Finally, the **most_common_frequencies** shows the frequency of each of the most common values, the nulls are 0.66% of the total values, and each of the other values represents 0.33% of the total values from the sample.

We can verify the information the next way:

```
pagila=# WITH total AS (SELECT count(*)::numeric cnt FROM address)
SELECT
  postal_code,  count(address.*),  round(count(address.*)::numeric  *
  100 / total.cnt, 2) AS “percentage”
FROM address, total
```

```
GROUP BY address.postal_code, total.cnt

HAVING count(address.*) > 1

ORDER BY 2 DESC;

 postal_code | count | percentage
-------------+-------+------------
             |     4 |       0.66
 22474       |     2 |       0.33
 52137       |     2 |       0.33
 9668        |     2 |       0.33
(4 rows)
```

We can see only the **null** and the other three **postal_code** values appear more than once in the whole table, appearing four times or twice, so excepting these, all the other values are distinct. Also, by doing some math (**count x 100 / total_rows**), we get the percentage the number of appearances represents versus the total number of values, that is the frequency.

The planner uses these and the other details from the **pg_statistics** catalog to resolve queries using the **WHERE** clause, so it can define the best path to retrieve just a fraction of the rows when required.

Statistics in pg_statistics_ext_data

So far, we have seen PostgreSQL keeps general statistics about the total number of rows per table, the space they occupy on the disk, and information about the selectivity per table column. However, there are cases where the queries correlate multiple different columns, and the single-column orientation of the regular statistics could be improved.

For such cases, PostgreSQL has the ability to compute *extended statistics*. These are a particular type of statistics and are not gathered by default, so the user or database administrator has to define them. The goal is to define the correlation between different columns, so the planner can improve its calculation when resolving queries. The way the data from different columns is related depends on the design of the database model and the tables, which is why human intervention is required.

We have to use the **CREATE STATISTICS** command to define the *extended statistics*. *Figure 13.1* shows the options for this command:

```
pagila=# \h CREATE STATISTICS
Command:     CREATE STATISTICS
Description: define extended statistics
Syntax:
CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    ON ( expression )
    FROM table_name

CREATE STATISTICS [ IF NOT EXISTS ] statistics_name
    [ ( statistics_kind [, ... ] ) ]
    ON { column_name | ( expression ) }, { column_name | ( expression ) } [, ...]
    FROM table_name

URL: https://www.postgresql.org/docs/14/sql-createstatistics.html
```

Figure 13.1: *CREATE STATISTICS command options.*

There are three types of extended statistics, each covering a distinct kind of column values correlation:

- Functional dependencies.
- Number of distinct values counts.
- Most common values list.

Functional dependencies

This kind of correlation is possible when the knowledge about one column is sufficient to determine another. This information lets the planner make accurate estimations about the total number of rows when the two columns are involved, for example, when using `AND`. Considering the same sample database as before, we can try this on the **`city`** table.

```
pagila=# CREATE STATISTICS extsts (dependencies) ON city_id, country_
id FROM city;

pagila=# ANALYZE city;

pagila=# SELECT

  stxname AS “ext_stats_name”,

  stxkeys AS “ext_stats_columns”,

  stxddependencies AS “stats_dependencies”

FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid)
```

```
WHERE stxname = 'extsts';

-[ RECORD 1 ]------+---------------------------------------
ext_stats_name     | extsts
ext_stats_columns  | 1 3
stats_dependencies | {"1 => 3": 1.000000, "3 => 1": 0.070000}
```

The **stats_dependencies** value shows that the column **city_id** (number 1) can 100% identify the **country_id** (number 3), whereas the other way, only 7%.

These estimations are beneficial, but they have some limitations. The planner assumes the dependencies on the columns are compatible and redundant, so it will do wrong estimations if they are incompatible. *Figure 13.2* shows how the planner will estimate 1 row even when the correlation is wrong **(NOTE: We will study EXPLAIN command later in this chapter).**

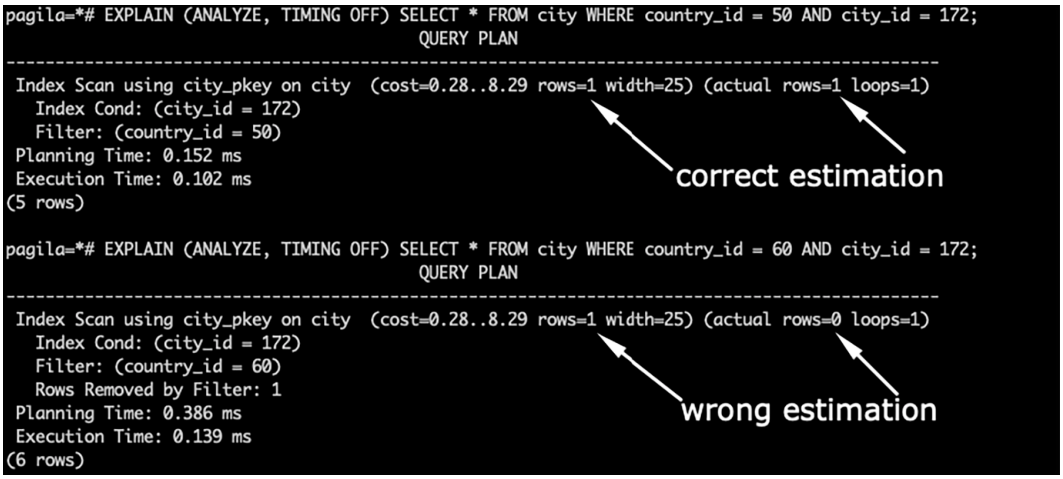

Figure 13.2: *Dependencies extended statistics, wrong estimation.*

Number of distinct values counts

The standard statistics rely on statical data for individual columns, which could lead to wrong estimations about the number of distinct values from a multiple columns combination.

The Number of Distinct values count (or N-Distinct) extended statistics can help when the queries request distinct values from combining different columns, for example, when using the `GROUP BY` clause.

We can try this type of statistic on the **pagila** example database. Let's use the **address** table. Imagine a set of queries selecting distinct values from the **district, city_id**, and **postal_code** columns; we might define the n-distinct extended statistics as follows.

```
pagila=# CREATE STATISTICS extsts2 (ndistinct) ON district, city_id,
postal_code FROM address;
pagila=# ANALYZE address;
pagila=# SELECT
  stxname AS "ext_stats_name",
  stxkeys AS "ext_stats_columns",
  stxdndistinct AS "stats_ndistinct"
FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid)
WHERE stxname = 'extsts2';


-[ RECORD 1 ]-----+-----------------------------------------------
------
ext_stats_name    | extsts2
ext_stats_columns | 4 5 6
stats_ndistinct   | {"4, 5": 601, "4, 6": 601, "5, 6": 601, "4, 5, 6":
601}
```

Consulting the information stored in the **pg_statistic_ext_data** catalog, we can verify the number of distinct values through the different **district** (number 4), **city_id** (number 5), and **postal_code** (number 6) column combinations. All of them produce 601 different values with the current data. *Figure 13.3* illustrates the planer does a correct estimation.

```
pagila=*# EXPLAIN (ANALYZE, TIMING OFF) SELECT COUNT(*) FROM address GROUP BY district, city_id, postal_code;
                                    QUERY PLAN
-------------------------------------------------------------------------------------
 HashAggregate  (cost=20.06..26.07 rows=601 width=26) (actual rows=601 loops=1)
   Group Key: district, city_id, postal_code
   Batches: 1  Memory Usage: 169kB
   ->  Seq Scan on address  (cost=0.00..14.03 rows=603 width=18) (actual rows=603 loops=1)
 Planning Time: 0.373 ms
 Execution Time: 1.057 ms
(6 rows)
```

Figure 13.3: *N-Distinct extended statistics.*

Most common values list

As we saw at the beginning of this subsection, the standard statistics stored in the **pg_statistics** catalog contain information about the most common values per column and their frequency. This information is good when filtering data from a table based on a single column, but this can lead to lousy planning when more columns are involved.

The Most common values list (MVC List) extended statistics can improve the planner's decisions when working with queries doing filtering on multiple column conditions. The following shows how to create this statistic on the `address` table.

```
pagila=# CREATE STATISTICS extsts3 (mcv) ON district, postal_code FROM
address;

pagila=# ANALYZE address;

pagila=# SELECT itms.*

  FROM pg_statistic_ext JOIN pg_statistic_ext_data ON (oid = stxoid),
  pg_mcv_list_items(stxdmcv) itms

  WHERE stxname = 'extsts3';


-[ RECORD 1 ]--+------------------------------

index          | 0

values         | {QLD,""}

nulls          | {f,f}

frequency      | 0.003316749585406302

base_frequency | 2.200165562458575e-05

-[ RECORD 2 ]--+------------------------------

index          | 1

values         | {Alberta,""}

nulls          | {f,f}
```

```
frequency      | 0.003316749585406302
base_frequency | 2.200165562458575e-05
-[ RECORD 3 ]--+-----------------------------
index          | 2
values         | {“”,65952}
nulls          | {f,f}
frequency      | 0.001658374792703151
base_frequency | 8.250620859219656e-06
-[ RECORD 4 ]--+-----------------------------
index          | 3
values         | {“Abu Dhabi”,41136}
nulls          | {f,f}
frequency      | 0.001658374792703151
base_frequency | 5.500413906146438e-06
...
```

Consulting the information stored for this extended statistics object, we can see the combination of **`values`**, their **`frequency`** as combined, and the **`base_frequency`**, which is the result computed as per-column frequency. For example, record number 4 shows the combination of **`district`** and **`postal_code`** **{“Abu Dhabi”,41136**} has a frequency of 0.16% and if the are computed per column, the frequency is only 0.0005%, which is a remarkable difference.

Explain plan

Before understanding explain plan, let’s review what the **`ANALYZE`** command is. As we saw in the previous section, the **`ANALYZE`** command gathers statistics about the contents of tables in the database and stores this metadata in the **`pg_class`**, **`pg_statistic`**, or **`pg_statistics_ext_data`** system catalogs. Eventually, the query planner uses this metadata which helps in determining the effective execution plans for queries.

EXPLAIN PLAN as the name suggests gives details about the queries' execution plan. It gives appropriate results when the statistics are updated. That is why it is recommended use **EXPLAIN PLAN** with **ANALYZE** command so that all metadata is up-to-date when the query plan is created.

Syntax as per PostgreSQL Documentation *(Reference: PostgreSQL Community Explain Plan).*

```
EXPLAIN [ ( option [, ...] ) ] statement

EXPLAIN [ ANALYZE ] [ VERBOSE ] statement


where option can be one of:


    ANALYZE [ boolean ]

    VERBOSE [ boolean ]

    COSTS [ boolean ]

    SETTINGS [ boolean ]

    BUFFERS [ boolean ]

    WAL [ boolean ]

    TIMING [ boolean ]

    SUMMARY [ boolean ]

    FORMAT { TEXT | XML | JSON | YAML }
```

The explain plan gives the details about whether it will use index scan or sequential scan, what plans are used to fetch the data if multiple tables are used, how many rows will come in the output of the query and many such details which can help in understanding the time taken by each step. This will help to tune to query in case there are performance issues.

Let's see the same example of one Department (**dept**) having multiple Employees in the **emp** table we used in Chapter 11 and Chapter 12. *Figure 13.4* shows the example of the join query for the **emp** and **dept** tables to fetch details of employees in **dept_id**

=2.

```
postgres=# EXPLAIN (ANALYZE,BUFFERS)
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                                    QUERY PLAN

-------------------------------------------------------------------------------------------------------------------
-----------
 Nested Loop  (cost=10000000000.13..10000000009.22 rows=1 width=668) (actual time=0.068..0.077 rows=2 loops=1)
   Buffers: shared hit=5
   ->  Seq Scan on emp e  (cost=10000000000.00..10000000001.06 rows=1 width=300) (actual time=0.034..0.037 rows=
2 loops=1)
         Filter: (dept_id = '2'::numeric)
         Rows Removed by Filter: 3
         Buffers: shared hit=1
   ->  Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368) (actual time=0.017..0.017 rows=1
loops=2)
         Index Cond: (dept_id = '2'::numeric)
         Buffers: shared hit=4
 Planning Time: 0.371 ms
 Execution Time: 0.149 ms
(11 rows)
```

Figure 13.4: *Example of EXPLAIN PLAN in a join query with ANALYZE and BUFFERS*

Notice that Sequential Scan is being used for the **`WHERE`** clause condition **e.dept_id =2**. There is a scope of tunning here where we can create an index on the Foreign Key column **`dept_id`** of the **`emp`** table. Let's create an index on this column and check the explain plan again.

```
CREATE INDEX INDX_EMP_DEPT_ID ON EMP(DEPT_ID);
```

Figure 13.5 shows the simple Explain Plan with no attributes. Be aware of the difference in the output of the explain plan in *Figures 13.4* and *13.5.*

Notice that Index Scan is used this time with the newly created index instead of Sequential Scan. This makes the performance of the query faster. We might not be able to see the difference when there are fewer records in the query output, but the increase in performance can be observed when the data increases.

```
postgres=# EXPLAIN
postgres-# SELECT *
postgres-# FROM EMP E,
postgres-#      DEPT D
postgres-# WHERE E.DEPT_ID = D.DEPT_ID
postgres-# AND E.DEPT_ID = 2;
                                    QUERY PLAN
-------------------------------------------------------------------------------
 Nested Loop  (cost=0.26..16.31 rows=1 width=668)
   ->  Index Scan using indx_emp_dept_id on emp e  (cost=0.13..8.15 rows=1 width=300)
         Index Cond: (dept_id = '2'::numeric)
   ->  Index Scan using dept_pk1 on dept d  (cost=0.13..8.15 rows=1 width=368)
         Index Cond: (dept_id = '2'::numeric)
(5 rows)
```

Figure 13.5: *Example of EXPLAIN PLAN in a join query without any attributes*

Below gives details of the useful parameters which should be kept in mind while tunning the queries and analyzing them using the explain plan. There are more configurations from the ones mentioned below, which could be checked using the **SHOW** command or in the **postgresql.conf** file, as we will see in the next section.

- **enable_indexscan (boolean)**
- **enable_nestloop (boolean)**
- **enable_seqscan (boolean)**
- **enable_sort (boolean)**

These are self-explanatory from the name of the parameter. By default, most of such parameters are on, and one can disable them depending upon the situation.

Best practices for the postgresql.conf parameters

When configuring a new PostgreSQL cluster or working with an existing one to improve performance, we can consider some standard parameter configurations widely advised as best practices for almost any system. These parameters are defined in the **postgresql.conf** file, generally located in the **$PGDATA** directory.

We need to remember that these best practices or baselines are generally effective and produce the desired results; however, there is always room for improvements, especially if the database model design or user workloads have some specifics. So, we can start with the best practices and develop a regular habit of monitoring our systems and tuning what is required over time.

The following is a list of the main configuration parameters we can tune based on best practices to bring good performance.

- **shared_buffers**
- **work_mem**
- **autovacuum**
- **effective_cache_size**
- **maintenance_work_mem**
- **max_connections**

In the previous chapters, we have seen a few of the above; now, we will review them and learn about their suggested initial values.

shared_buffers

This parameter defines the size of the memory area named the same, which we also might know as the "database cache." This memory area will keep the most acceded rows so new reads can retrieve them from here rather than going to disk; also, all the data changes are written in this area instead of immediately on disk.

The default value of this parameter is very conservative, just 128 MB. The best practice for this parameter is to set it between 15% and 25% of the total RAM. So, for example, in a 64 GB RAM system, the **`shared_buffers`** should be set to 16 GB. Changing this parameter demands a server restart.

work_mem

This parameter determines the size of the memory area named the same. Differently from the **`shared_buffers`**, which exists just one for the whole database cluster (shared by all the existing databases), a **`work_mem`** is allocated per user session. This area is used for all the sort operations, such as **`ORDER BY, DISTINCT`**, and **`MERGE JOINS`**.

If the operation using the **`work_mem`** requires more space to complete, then PostgreSQL will use temporary files written on disk. Tuning this parameter to avoid the IO disk operations can improve performance.

The default value of this parameter is just 4 MB. We must consider the available resources and the number of concurrent user sessions to adjust it.

To be on the safe side, we can pick the **`max_sessions`** value, which also should be sized right. So, the calculation for this parameter can be expressed the following way: **`work_mem = 25% of total RAM / max_connextions.`**

We can change this parameter without a server restart and even set it at the role (user) level using the **`ALTER USER`** command. *(Reference: PostgreSQL Community alter user).*

autovacuum

This parameter controls if the autovacuum background process is enabled or not. As we have seen in previous chapters, vacuuming the tables and indexes is vital in a PostgreSQL system. By default, this parameter is enabled, so the recommendation is to keep it this way.

If, for any reason, you need to disable the autovacuum, the recommendation is to disable it by table rather than for the whole cluster, changing it at the cluster level

in the postgresql.conf file requires a server restart. You can use the **ALTER TABLE** command to disable the autovacuum at the table level. *(Reference: PostgreSQL Community alter table).*

effective_cache_size

This parameter lets the query planner know how much memory is expected to be available in the system for disk caching within the database. It does not represent an allocated memory area, but the planner uses its value to decide whether the operations to resolve specific queries will fit the RAM.

If the value is too low, the planner might disregard the index scans and prefer sequential table scans, which usually perform slower to access specific data subsets.

The default for this parameter is 4 GB, and the best practice is setting it between 50% and 75% of the total RAM. Change it doesn't require a server restart.

maintenance_work_mem

This parameter defines the size for the memory area named the same, which is used for the maintenance tasks **VACUUM, CREATE INDEX**, and **ALTER TABLE ADD FOREIGN KEY**. Since only one of these tasks can be running simultaneously per connection and are usually executed in a lower proportion than the **work_mem** operations, its value can be significantly larger than **work_mem.**

Its default value is 64 MB, which is very conservative. The advisable value for this parameter is between 5% and 10% of the total RAM.

max_connections

This parameter limits the maximum number of simultaneous user database connections. As we saw before, each of these connections can hold a **work_mem** size memory area at least.

Remember, PostgreSQL has a process-based architecture, so each user session represents a new process at the database server. There is some relation between the number of CPUs available in the system and the maximum number of processes it can handle efficiently.

The default of this parameter is 100, and usually, it is recommended to set this value lower than 10 per CPU and consider 20 per CPU as the limit. If you need to allow more user sessions than these, consider a pooler such as **pgbouncer**.

Summary

The above parameters are the main ones we advise you to verify when working to improve the performance of your system. Next is a summary table with the default and suggested values.

Parameter	Default values	Best practice value
`shared_buffers`	128 MB	Between 15 - 25% of total RAM
`work_mem`	4 MB	25% of total RAM / max_connextions
`autovacuum`	ON	ON
`effective_cache_size`	4 GB	Between 50% - 75% of total RAM
`maintenance_work_mem`	64 MB	Between 5% - 10% of total RAM
`max_connections`	100	Up to 20 per CPU

Conclusion

In this chapter, we have seen the concepts which can help to increase the performance of the database and helps in tunning the queries. The topics like Index creation and its maintenance, statistics, and explain plan are important from the DBA perspective. We also discussed the best practices to be considered in one of the most important `postgresql.conf` file.

In the next chapter, we will discuss how-to, tips, and tricks to troubleshoot the database.

Bibliography

- Devrim Gündüz pagila: **https://github.com/devrimgunduz/pagila**
- PostgreSQL Community pg_stats: **https://www.postgresql.org/docs/14/view-pg-stats.html**
- PostgreSQL Community Index: **https://www.postgresql.org/docs/current/sql-createindex.html**
- PostgreSQL Community Re Index: **https://www.postgresql.org/docs/current/sql-reindex.html**
- PostgreSQL Community Index Types: **https://www.postgresql.org/docs/current/indexes-types.html**

- PostgreSQL Community Explain Plan: **https://www.postgresql.org/docs/14/sql-explain.html**
- PostgreSQL Community alter user: **https://www.postgresql.org/docs/14/sql-alteruser.html**
- PostgreSQL Community alter table: **https://www.postgresql.org/docs/14/sql-altertable.html**

Chapter 14
Troubleshooting

Introduction

All technological systems are susceptible to unexpected errors and failures; even in the most stable workloads, the possibility of hitting a hidden bug or an unknown situation is present. While working with PostgreSQL, we can face this kind of situation, and it is essential to know the tools and methods we can use to troubleshoot and debug the errors to avoid or fix them.

In previous chapters, we reviewed the features of PostgreSQL and how to configure them and tune our systems. In this chapter, we will learn about the principal methods and tools we can use when the time to debug an error comes in. We will study these concepts from the PostgreSQL and Operating System points of view.

Structure

This chapter includes the following sections.

- Debugging using log files.
- Debugging using PostgreSQL tools and commands.
- Debugging using Operating System tools and commands.

Objectives

Completing this chapter, you will gain knowledge of the main techniques and tools to troubleshoot an unexpected error. Also, you will be familiar with the relationship between the Operating System and the PostgreSQL processes.

Debugging using log files

Like any other system, PostgreSQL can log helpful information in its log files. The logged data can help understand the sequence of events and their relationship. Also, some information is not visible from the DB but from the log files.

What is logged and where is logged depend on a set of configuration parameters in the **`postgresql.conf`** file. Depending on the system workload and the type of operations, we might need some details to be logged, and others do not. However, we can consider some standard and usually recommended logging parameters. The following subsections describe the main parameters we should set for logging valuable information.

Where to log

It is important to define where the log files will be stored; this way, we will ensure their availability when needed and control how to recycle them to avoid running out of space. Also, under high workloads, avoid causing an impact on the disk for the database operations by moving the log files to a different disk. The following are the principal parameter we can set to define these aspects for the log files.

logging_collector

When enabling this parameter, a background process is started, and it captures all the messages sent to the standard error (**`stderr`**) and redirects them into log files. The default value for this parameter is **off**; however, it is highly recommended to set it to **on**.

Even when it is possible to log the PostgreSQL messages without enabling the **`logging_collector`** parameter, the only way to ensure all the messages will be captured, and there will not be any loss is by enabling this parameter. Also, this process adds support for the rotation capabilities.

log_destination

PostgreSQL supports a variety of destinations for the log messages, which adds flexibility to decide what is the best depending on the infrastructure and solution design. The possible values for this parameter are: **`stderr, csvlog,`** and **`syslog`**. When running PostgreSQL in Windows, it is possible to set this parameter to **`eventlog`**.

The **`stderr`** is the default value, and it will generate the output in a text format. The **`csvlog`** will format the output to be separated by commas, which is especially useful when we use the log messages to feed other different applications or analyzer systems. To use the **`syslog`** (Linux/Unix-alike systems) or the **`eventlog`** options, it is necessary to configure the operating system accordingly so the messages are processed as expected. Configuring these operating system utilities is beyond the scope of this book, so reviewing them, is worth it.

log_directory

As you might already deduce, this parameter controls the directory path where the log files will be created and stored. This parameter only becomes valid when the **`logging_collector`** is enabled.

You can set the value for this parameter as an absolute path or relative to the cluster data directory, a.k.a **`$PGDATA`**. It is default set to **`log`**, which means the log files will be created in the **`$PGDATA/log`** directory. It is highly recommended, especially if your system handles a high workload, to set this directory in separate storage from the one where the database data is. *Figure 14.1* shows a system with this parameter configured in a different storage from the database data.

```
postgres=# show data_directory;
       data_directory
----------------------------
 /var/lib/postgresql/14/main
(1 row)

postgres=# show log_directory;
 log_directory
---------------
 /pg/logs
(1 row)
```

Figure 14.1: *log_directory in a separate path from the database data*

log_filename

This parameter can be used when the **logging_collector** is enabled, and it sets the file name convention to be used when creating the log files. This parameter supports **strftime** patterns to produce time-varying filenames. *(Reference: strftime).*

The default value is **postgresql-%Y-%m-%d_%H%M%S.log**, so considering the file creation date as **Feb 15, 2023, 17:22:45**, the filename will be **postgresql-2023-02-15_172245.log.** We can define the log file names in a way it we do not need an external scheduled job to remove the old files; for example, if we set the file names to be set to the day of the week or month, and have the **log_truncate_on_rotation** enabled, every time the *same* logfile gets created a new one will take the place of the previous overwriting it. This will consistently keep the same number of files in place.

log_rotation_age

This parameter controls when to rotate the log file to a new one based on time; this is possible only when the **logging_collector** is enabled. The value is expressed in minutes; the default value is 24 hours (1440 minutes), so a new log file is added at the beginning of a new day. It is possible to set this parameter to 0 (zero) to deactivate the time-based rotation mechanism.

log_rotation_size

Similarly to the last parameter, this one defines when to create a new log file based on the file size when the **logging_collector** is enabled. If the value has no units specified, it is calculated as kilobytes. The default value is 10MB, and setting it to 0 (zero) disables the size-based rotation.

log_truncate_on_rotation

When the **logging_collector** is enabled, this parameter controls if the messages sent to an existing log file will be appended (**off**) or the file will be truncated/overwritten (**on**). As we saw before, this can be combined with the **log_filename** to control the number of files to store.

For example, if the **log_filename** is configured as **postgresql-%H.log**, the **log_rotation_age** is on its default of **1440** minutes (1 day), and the **log_truncate_on_rotation** is **on**, then every hour, a new empty file will be created, and PostgreSQL will keep only 24 files.

What to log

Setting the PostgreSQL logger correctly will provide valuable information we can use when we need to debug system operations and events. The following are the main options we should consider configuring, but they depend on the type of workload, so we need to evaluate them to avoid extra logging that might not add value.

log_line_prefix

This is one of the most useful configuration options; whenever we aim to analyze the database logs manually or use a parser such as **`pgbadger`**, we should consider configuring this parameter with all the relevant information. This parameter defines a **`printf-style`** string that will be added at the beginning of each log line. The default value is **`%m [%p]`**. *(Reference: PostgreSQL Community **`log_line_prefix`**)*

Among others, the information we can include by setting this parameter might contain:

- The specific timestamp (%t).
- The process ID (%p).
- The PostgreSQL database (%d).
- The client database user (%u), application name (%a), and/or source host (%h).

These details will help to identify the process, user, and source for a specific log event, making the analysis and troubleshooting easier. Let us say you have the following configuration:

```
postgres=# SHOW log_line_prefix;
             log_line_prefix
-----------------------------------------
 %t [%p]: db=%d user=%u,app=%a,client=%h
(1 row)
```

And there is some error when calling an unexisting function:

```
postgres=# SELECT * FROM new();
ERROR:  function new() does not exist
```

You will see the following message as illustrated in *Figure 14.2*:

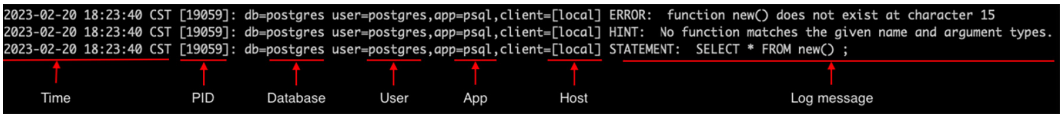

Figure 14.2: *Example of output as per log_line_prefix setting*

log_connections/log_disconnections

As the name suggests, these parameters will add a log event for every user connection and/or disconnection, depending on how we configure them. The default value is **off** for both of them.

log_min_duration_statement

This parameter controls when the completion of the queries' statements will be logged; a new log message will be added when the execution of a given statement is longer than the time defined in this parameter. The used unit is milliseconds, and the default value is **-1**, meaning this option is disabled. A 0 (zero) value will cause all the statements to be logged.

This is one of the most useful logging options.

log_lock_waits

When enabled, a log message will be added if a session waits longer than the defined time in the **deadlock_timeout** parameter (default 1 second) to acquire a lock. The default value is **off**. The log messages produced by this setting help to identify if the locking waits are causing issues.

log_autovacuum_min_duration

This parameter defines to log of any autovacuum events running at least this amount of time. The units are milliseconds, and the default value is **-1**, which disables this option. Setting a 0 (zero) value causes all the autovacuum operations to be logged. By defining a meaningful amount of time, we can get insight if the autovacuum is causing performance or blocking issues.

A well-configured logger in PostgreSQL will produce a great tool for debugging errors and failure situations. Even when it is possible to read and gather details from the postgres logs manually, it is highly recommended to parse and analyze the log files with some other tool, for example, **pgbadger,** which we saw in *Chapter 10*.

Parameters summary

As we saw, multiple configuration parameters can help to log relevant information we can use in case of troubleshooting. *Table 14.1* summarize the reviewed parameter and the suggested configuration.

Parameter	Default value	Recommended value	Units
logging_collector	off	on	
log_destination	stderr	stderr, csvlog	
log_directory	log	A different storage path from the DB data.	
log_filename	postgresql-%Y-%m-%d_%H%M%S.log	postgresql-%a.log	
log_rotation_age	1440	1440	minutes
log_rotation_size	'10MB'	0	KB, MB, GB
log_truncate_on_rotation	off	on	
log_line_prefix	%m [%p]	%t [%p]: db=%d user=%u, app=%a,client=%h	
log_connections	off	on	
log_disconnections	off	on	
log_min_duration_statement	-1	500	milliseconds
log_lock_waits	off	on	
log_autovacuum_min_duration	-1	1000	milliseconds

Table 14.1: *Main configuration parameters for PostgreSQL logging.*

Debugging using PostgreSQL tools and commands

As we have studied before in this book, PostgreSQL is composed of a set of background processes responsible for different tasks, and there are a few native commands we can use to instruct them. Also, the system maintains a variety of catalogs that contain

information we can use when we need to troubleshoot. So, while accessing postgres with the right privileges, we can execute the following commands to gather details to solve some issues.

In the following paragraphs, we will study the main and basic commands or queries we can use to gather details about *what is happening* in the database system. Then a few other commands to instruct the system to execute certain actions.

Gather information

Before trying to fix something, we must know what is happening, so when troubleshooting PostgreSQL, we can use the following to get insights.

Check the PostgreSQL version

It might look basic, but when we start troubleshooting an issue, the initial step we can do is verify the specific version the system is running. With this information, we can identify if the version is affected by an identified bug and if the case, if there is a fix already, if the version supports some feature that helps to solve the issue, and if it is still under support.

```
postgres=# SELECT version();
```

Check database and objects size

There are situations where the issue we are facing is related to the disk running out of free space; in such cases, knowing the size of the database and the tables and indexes comes in handy.

We can get the database size in a human-readable way with the following:

```
postgres=# SELECT pg_size_pretty(pg_database_size('database_name'));
```

Then, to be more specific, we can verify the total size a table is consuming in the disk, including the data and the indexes:

```
postgres=# SELECT

   pg_size_pretty(pg_total_relation_size('table_name'));
```

If we want to know the size of the table data and the indexes separately, we can use the following:

```
postgres=# SELECT
```

```
    pg_size_pretty(pg_table_size('table_name')) AS "Table Size",
    pg_size_pretty(pg_indexes_size('table_name')) AS "Indexes Size";
```

Check database connections

We can also know the number of database connections and their state, giving us insight into what the database servers.

```
postgres=#  SELECT COUNT(*) AS "# of Sessions", state AS "State" FROM
pg_stat_activity GROUP BY state;
```

If required, we can add some extra details, for example, the following. We can know the database name and database user alongside the last details:

```
postgres=#  SELECT
   datname AS "Database",
   usename AS "User",
   state AS "State",
   COUNT(*) AS "# of Sessions"
FROM pg_stat_activity
GROUP BY state, datname, usename
ORDER BY "Database", "User", "# of Sessions";
```

Finally, we might want to know for how long these sessions are being connected and for how long their queries have been running:

```
postgres=#  SELECT
   datname AS "Database",
   usename AS "User",
   state AS "State",
   now() - backend_start AS "Session Time",
   now() - query_start AS "Query Time"
FROM pg_stat_activity
ORDER BY "Database", "User", "Session Time";
```

Figure 14.3 illustrates the output from the last command:

```
postgres=# SELECT
   datname AS "Database",
   usename AS "User",
   state AS "State",
   now() - backend_start AS "Session Time",
   now() - query_start AS "Query Time"
FROM pg_stat_activity
ORDER BY "Database", "User", "Session Time";
 Database |     User      | State  |     Session Time      | Query Time
----------+---------------+--------+-----------------------+------------
 postgres | postgres      | active | 00:00:16.536177       | 00:00:00
 sumo     | postgres      | idle   | 00:00:46.058131       |
```

Figure 14.3: *Checking user session details*

Check slow queries

The following queries are handy for quickly identifying queries that might have been bad-behaving and causing issues. To use the following, we need to have added the **`pg_statements`** extension, which we learned in *Chapter 10: Most used Extentions/ Tools* ; you can review it for a few extra queries.

The following will return the queries that have taken longer than 5 seconds to complete and their execution time:

```
postgres=#  SELECT query, total_time
FROM pg_stat_statements
WHERE total_time > 5000 ORDER BY total_time DESC;
```

Check statistics

In the previous chapter, we studied the statistics the query planner uses to decide the quickest path to get the data, but also we mentioned some other kinds of statistics PostgreSQL collects regarding the usage and state of the database and its objects. These statistics can help us understand how the system behaves and identify issues.

The statistics are stored in multiple **`pg_stat*`** views; the following are some of the most relevant.

We can verify the last time the tables were vacuumed and analyzed. Having tables with long periods without vacuum or analysis on them can point to a potential performance issue and, in some cases, affect the whole system.

```
postgres=#  SELECT
   schemaname AS “Schema”, relname AS “Table”,
```

```
    last_vacuum, last_autovacuum, last_analyze, last_autoanalyze
FROM pg_stat_user_tables ORDER BY last_vacuum DESC;
```

We can use the following query to get an insight into how the tables are accessed. If the number of sequential scans exceeds the index scans, the system might face some latency and slowness.

```
postgres=#  SELECT
    schemaname AS “Schema”, relname AS “Table”,
    seq_scan AS “# Seq Scan”, idx_scan AS “# Index Scan”
FROM pg_stat_user_tables ORDER BY “# Seq Scan” DESC;
```

When our system includes a replication setup, verifying if the replicas are synchronized and not getting behind by too far from the changes in the primary node is valuable. We can use the following in the primary to know the status of the replicas.

```
postgres=#  SELECT
    application_name AS “Client App Name”, state AS “State”,
    pg_current_wal_lsn() AS “Current WAL Position”,
    replay_lsn AS “Replayed WAL Position”,
    replay_lag AS “Replay Lag”,
    pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),replay_lsn)) AS
    “Lag Size”
FROM pg_stat_replication ORDER BY “Lag Size”  DESC;
```

Figure 14.4 illustrates a replication setup with two replicas in good shape:

```
postgres=# SELECT
postgres-#    application_name AS "Client App Name",
postgres-#    state AS "State",
postgres-#    pg_current_wal_lsn() AS "Current WAL Position",
postgres-#    replay_lsn AS "Replayed WAL Position",
postgres-#    replay_lag AS "Replay Lag",
postgres-#    pg_size_pretty(pg_wal_lsn_diff(pg_current_wal_lsn(),replay_lsn)) AS "Lag Size"
postgres-# FROM pg_stat_replication ORDER BY "Lag Size"  DESC;
 Client App Name |   State   | Current WAL Position | Replayed WAL Position |   Replay Lag    | Lag Size
-----------------+-----------+----------------------+-----------------------+-----------------+----------
 replica_2       | streaming | D077/17215018        | D077/17215018         | 00:00:00.047488 | 0 bytes
 replica_1       | streaming | D077/17215018        | D077/17215018         | 00:00:00.009076 | 0 bytes
(2 rows)
```

Figure 14.4: *Checking replication health with pg_stat_replication*

Instruct PostgreSQL

In some previous chapters we have studied some special commands to execute specific actions in PostgreSQL, apart from querying the tables data. When troubleshooting an issue, sometimes we need to perform corrective actions, the following commands can help.

Vacuuming and analyzing

As you can relate already, there are special commands to execute the VACUUM and/or ANALYZE operations on demand. They can help if we have identified the cause of an issue as the absence of the vacuum cleaning or the freshness of the statistics is outdated. The commands to execute such tasks are kind of obvious:

```
postgres=#  VACUUM [tablename];


postgres=#  ANALYZE [tablename];
```

In both cases, we can include the name of a specific table to execute the corresponding routine just in that object; otherwise, the execution will affect the whole database. Review *Chapter 6: PostgreSQL Internals to* refresh some details and options about **`VACUUM`**.

Terminate queries or user sessions

Under certain circumstances, you might need to terminate one or more user sessions connected to the database. This can be because of an undesired execution or because the identified sessions are running bad queries and affecting all the other users by slowing down the system or blocking access to the tables.

Two options to do this:

```
-- Terminate a query but keep the connection alive.

postgres=# SELECT pg_cancel_backend(pid);


-- Terminate a query and kill the connection.

postgres=# SELECT pg_terminate_backend(pid);
```

The **`pid`** is the process identifier of the user session; this detail can be found in the **`pg_stat_activity`** catalog we saw before. We can even terminate *all* the sessions for a specific database user if we require it:

```
postgres=#  SELECT pg_terminate_backend(pid)
  FROM pg_stat_activity
  WHERE usename = 'database_username';
```

Manage replication

There are some special cases when we need to manage the state of the existing replication setup, for example pausing the replica, resuming it and verifying the role of a given server. The following commands come in handy.

To verify the role of a server in the replication setup, we can check if the server is *in recovery* mode, if **true** means it is a replica, **false** it is the primary server.

```
postgres=#  SELECT pg_is_in_recovery()::text;
```

We can also pause the replication and then resume it again when required. Be aware that pausing an ongoing replica will cause the primary to retain WAL files, which can cause storage issues if the disk space is limited.

```
-- Check the replication status.
postgres=#  SELECT pg_get_wal_replay_pause_state();

-- Pause an ongoing replication.
postgres=#  SELECT pg_wal_replay_pause();

-- Resume a paused replication.
postgres=#  SELECT pg_wal_replay_resume();
```

Under certain circumstances, you might need to change the role of one of the replica servers to turn it into a new primary, for example, if the current primary server becomes unavailable. In this situation, you need to handle how the database users are connecting and how to manage the former primary server rejoin. For now, these details are out of the scope of this book, but it is advisable to learn about them.

```
-- While connected to a replica server.
postgres=#  SELECT pg_promote();
```

You can refer to *Chapter 8, Replicating Data*, to review some extra information about replication in PostgreSQL.

Debugging using Operating System tools and commands

When facing issues with our PostgreSQL system, we can collect relevant information from the Operating System itself. Ultimately, all the processes composing the PostgreSQL service and the user sessions run at the Operating System level.

Understanding the Operating Systems layers, components, and services is a powerful tool for any Database Administrator. In this book, we are not going in deep into all the OS concepts; however, we can study some of the basic and main tools we can use to troubleshoot our systems. In this chapter, we are considering Linux-alike Operating Systems, so we are not including tools and examples for Windows platforms.

We have a large set of tools and commands from the Operating System when we need to troubleshoot, some can help to get a global view of the running services and system-wide resources, and others will give insight about specific processes, and we also have a couple of options to get log entries or system events so that we can review them over the time.

Service and system-wide tools

When we troubleshoot an issue, we can start by checking the general status of the service and trying to identify if the system is showing some unexpected behavior from the resource consumption point of view.

systemctl

When we install PostgreSQL on different Linux flavors, we will get a service unit, and the recommendation is to handle the postgres service through it. We can verify the status of a service, stop, start, or restart it if required.

```
-- To verify the status of the PostgreSQL service

root@:~# systemctl status postgresql


-- To stop the PostgreSQL service

root@:~# systemctl stop postgresql


-- To start the PostgreSQL service
```

```
root@:~# systemctl start postgresql


-- To restart the PostgreSQL service

root@:~# systemctl restart postgresql
```

Figure 14.5 illustrates the output of the **`systemctl`** command when checking the status of the PostgreSQL service, note, in this case, the service has been active since four days. Also, the last lines show the latest messages from the postgres log file, so we can quickly see if something is happening:

```
root@ubuntu-focal:~# systemctl status postgresql
● postgresql.service - PostgreSQL RDBMS
     Loaded: loaded (/lib/systemd/system/postgresql.service; enabled; vendor preset: enabled)
     Active: active (exited) since Sun 2023-02-19 23:23:45 UTC; 4 days ago
   Main PID: 984 (code=exited, status=0/SUCCESS)
      Tasks: 0 (limit: 1131)
     Memory: 0B
     CGroup: /system.slice/postgresql.service

Feb 19 23:23:45 ubuntu-focal systemd[1]: Starting PostgreSQL RDBMS...
Feb 19 23:23:45 ubuntu-focal systemd[1]: Finished PostgreSQL RDBMS.
```

Figure 14.5: *Checking the PostgreSQL service status with* `systemctl`.

free

The **`free`** command will show information about the system memory; we get details about the total memory, what is used and free, and if the swap (an auxiliary memory space on disk) is being used. *(Reference: Linux Manual free)*

The RAM is a limited resource and one of the most critical for a database system, so running out of free memory will have an impact. *Figure 14.6* shows an output from the **`free`** command.

```
-- Getting memory details in human-readable format

root@:~# free -h
```

```
root@ubuntu-focal:~# free -h
               total        used        free      shared  buff/cache   available
Mem:           976Mi       174Mi       233Mi        14Mi       568Mi       629Mi
Swap:            0B          0B          0B
```

Figure 14.6: *Output from the free command in a human-readable format.*

df

The **df** command is a tool to get information about the space on the mounted filesystems. We can get an insight into whether any filesystems are running out of free space. *Figure 14.7* illustrates the output from the command **df**. *(Reference: Linux Manual df)*

```
-- Getting filesystem space usage

root@:~# df -Ph
```

```
root@ubuntu-focal:~# df -Ph
Filesystem      Size  Used Avail Use% Mounted on
udev            472M     0  472M   0% /dev
tmpfs            98M  1.1M   97M   2% /run
/dev/sda1        39G  4.1G   35G  11% /
tmpfs           489M   28K  489M   1% /dev/shm
tmpfs           5.0M     0  5.0M   0% /run/lock
tmpfs           489M     0  489M   0% /sys/fs/cgroup
/dev/loop0       64M   64M     0 100% /snap/core20/1778
/dev/loop2       68M   68M     0 100% /snap/lxd/22753
/dev/loop3       50M   50M     0 100% /snap/snapd/17883
/dev/loop4       92M   92M     0 100% /snap/lxd/24061
vagrant         466G  383G   84G  83% /vagrant
/dev/loop5       50M   50M     0 100% /snap/snapd/18357
/dev/loop6       64M   64M     0 100% /snap/core20/1822
tmpfs            98M     0   98M   0% /run/user/1000
```

Figure 14.7: *Output from the df command*

Processes-oriented tools

Additional to the previous tools, we will review a couple oriented to processes. With these, we can review information about specific processes running in the system.

top/htop

When looking at a system facing issues, we can use the **top** tool to get information about the different processes running, and we can sort them per the most demanding processes, which are using more CPU and memory or causing the highest load. The **htop** variant includes a more visual interface and supports some actions through clicks. *Figure 14.8* illustrates a **htop** output:

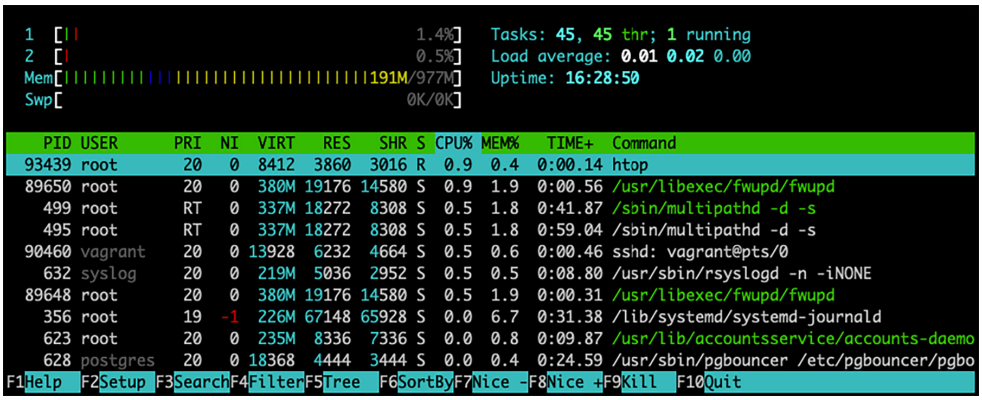

Figure 14.8: *An htop command output*

ps/pgrep

The **ps** and **pgrep** commands help to list and find a certain running process. We can verify if the expected processes are under execution or if any other that should not be running actually exists. For example, we can verify if all the PostgreSQL background processes are actually running.

```
-- Getting a list of the postgres processes with ps
root@:~# ps -fea | grep postgres

-- Getting the same list pgrep
root@:~# pgrep -a postgres
```

Figure 14.9 illustrates how to get a list of the postgres related processes using both commands:

```
root@ubuntu-focal:~# ps -fea | grep postgres
postgres     628       1  0 Feb23 ?        00:00:25 /usr/sbin/pgbouncer /etc/pgbouncer/pgbouncer.ini
postgres   90694       1  0 00:06 ?        00:00:00 /usr/lib/postgresql/14/bin/postgres -D /var/lib/postgresql/14/main -c config
_file=/etc/postgresql/14/main/postgresql.conf
postgres   90695   90694  0 00:06 ?        00:00:00 postgres: 14/main: logger
postgres   90697   90694  0 00:06 ?        00:00:00 postgres: 14/main: checkpointer
postgres   90698   90694  0 00:06 ?        00:00:00 postgres: 14/main: background writer
postgres   90699   90694  0 00:06 ?        00:00:00 postgres: 14/main: walwriter
postgres   90700   90694  0 00:06 ?        00:00:00 postgres: 14/main: autovacuum launcher
postgres   90701   90694  0 00:06 ?        00:00:00 postgres: 14/main: stats collector
postgres   90702   90694  0 00:06 ?        00:00:00 postgres: 14/main: logical replication launcher
root       93914   90477  0 01:10 pts/0    00:00:00 grep --color=auto postgres
root@ubuntu-focal:~#
root@ubuntu-focal:~#
root@ubuntu-focal:~# pgrep -a postgres
90694 /usr/lib/postgresql/14/bin/postgres -D /var/lib/postgresql/14/main -c config_file=/etc/postgresql/14/main/postgresql.conf
90695 postgres: 14/main: logger
90697 postgres: 14/main: checkpointer
90698 postgres: 14/main: background writer
90699 postgres: 14/main: walwriter
90700 postgres: 14/main: autovacuum launcher
90701 postgres: 14/main: stats collector
90702 postgres: 14/main: logical replication launcher
```

Figure 14.9: *Listing the postgres background processes with* **ps** *and* **pgrep** *commands*

Log and events

As we saw in the first section of this chapter, having a good PostgreSQL logger to keep track of the events happening in the system is a helpful tool. In the Operating System, some options exist to get information similarly. We can review historical events and verify their sequence of relevant messages, so when debugging an issue, we can better understand what happened.

journalctl

This tool will show us the log entries from the **systemd** journal; the **systemd** is the component of the Operating System in charge of handling the running services. As we saw a few paragraphs above, we can use the **systemctl** command to verify the current state of a service and stop, start, or restart it. The **journalctl** will help us review the services' historical log entries. *(Reference: Linux Manuel journalctl)*

Access to the messages thrown during the event always comes in handy when troubleshooting. The following are a couple of options we can do with **journalctl**.

```
-- Get all the available journal entries

root@:~# journalctl


-- Get all the available journal entries for a specific service

-- (postgresql)

root@:~# journalctl -u postgresql


-- Get the journal entries for a specific service (postgresql)

-- for the last 5 minutes

root@:~# journalctl -u postgresql --since “5 minutes ago”
```

Figure 14.10 shows the output of the last command example.

```
root@ubuntu-focal:~# journalctl -u postgresql@14-main.service --since "5 minutes ago"
-- Logs begin at Thu 2022-12-15 00:09:17 UTC, end at Fri 2023-02-24 22:37:17 UTC. --
Feb 24 22:36:26 ubuntu-focal systemd[1]: Stopping PostgreSQL Cluster 14-main...
Feb 24 22:36:26 ubuntu-focal systemd[1]: postgresql@14-main.service: Succeeded.
Feb 24 22:36:26 ubuntu-focal systemd[1]: Stopped PostgreSQL Cluster 14-main.
Feb 24 22:36:26 ubuntu-focal systemd[1]: Starting PostgreSQL Cluster 14-main...
Feb 24 22:36:29 ubuntu-focal systemd[1]: Started PostgreSQL Cluster 14-main.
```

Figure 14.10: *Output of journalctl for the postgresql service in the last 5 minutes*

Conclusion

As a system administrator, in our case a Database Administrator, we will face situations with unexpected and never seen errors. Being able to find the cause and provide a fix is one of the more relevant responsibilities of the role.

The good thing, there are plenty of tools and techniques to debug and troubleshoot the issues. In this chapter, we have learned a few from the database and operating system points of view. We can use some of the studied queries and commands to get valuable information. And also, we reviewed the log files and event messages are a helpful resource, so knowing how to configure them and access them is essential.

In the next and final chapter, we will learn about the PostgreSQL Community and the importance of the member's contributions, and how we can be part of it.

Bibliography

- PostgreSQL Community log_line_prefix: **https://www.postgresql.org/docs/14/runtime-config-logging.html#GUC-LOG-LINE-PREFIX**
- Linux Manual free: **https://man7.org/linux/man-pages/man1/free.1.html**
- Linux Manual df: **https://man7.org/linux/man-pages/man1/df.1.html**
- Linux Manual journalctl: **https://man7.org/linux/man-pages/man1/journalctl.1.html**
- strftime: **https://strftime.org/**

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Chapter 15
Contributing to PostgreSQL Community

Introduction

In the previous chapters, we have seen multiple PostgreSQL concepts and how to use, configure, and manage them from the operational point of view. This chapter will cover how to contribute to the community and a basic understanding of how the PostgreSQL Community works.

Structure

This chapter includes the following sections:

- PostgreSQL Community and its structure
  - Core Members of the PostgreSQL Community
  - Working pattern of PostgreSQL Community
  - Earning of PostgreSQL Community
- Different ways to contribute
  - Code contributor
  - Bug reporter
  - Participate in the mailing lists

- Improving or creating new documentation
- Participating in events
- Supporting the community

Objectives

The main objective of this chapter is to understand how PostgreSQL Community works and how one can contribute to improving the software and services.

PostgreSQL community and its members

Before understanding how to contribute to PostgreSQL, let us first try to understand the PostgreSQL Community and its functioning briefly in the next few sections.

Core members of the PostgreSQL community

The PostgreSQL community is made up of many contributors and developers, but there is a small group of core members who are responsible for the overall direction and management of the project. The core members of the PostgreSQL project are:

Tom Lane: Tom Lane is one of the longest-standing members of the PostgreSQL community and has significantly contributed to the project over the years. He is involved in all aspects of PostgreSQL, including new features, performance improvements, and bug evaluating and fixing. *(Reference: Tom Lane)*

Bruce Momjian: Bruce Momjian is another long-standing PostgreSQL community member, co-founder and core team member of the PostgreSQL Global Development Group. He has made significant contributions to the project, including writing much of the original documentation and helping to develop key features such as the PL/pgSQL procedural language.

Robert Haas: Robert Haas is a prominent member of the PostgreSQL community and a core development team member. He has made significant contributions to the project, including developing several key features such as parallel query execution, EXPLAIN PLAN in XML/JSON output, backup compression, backup targets, backup manifests, and many such features.

Magnus Hagander: Magnus Hagander is a core member of the PostgreSQL development team and is responsible for the PostgreSQL infrastructure to keep the services of postgresql.org up and running, including the PostgreSQL build farm, website, and mailing lists.

Peter Eisentraut: Peter Eisentraut is a long-standing member of the PostgreSQL community and a core development team member. He has made significant contributions to the project and has developed several popular PostgreSQL extensions, including pgAdmin, the PostgreSQL JDBC driver, and the PostgreSQL ODBC driver. These extensions have helped to enhance the functionality and usability of PostgreSQL and have made it easier for developers to work with the software.

These are few among many core team members of the PostgreSQL Community, and mentioning every member is outside the scope of this book. Overall, the core members of the PostgreSQL project are responsible for managing the overall direction and development of the software and play a key role in ensuring the quality and reliability of the PostgreSQL database management system.

Working pattern of PostgreSQL community

The PostgreSQL Community is an open and collaborative community that works together to develop and maintain the PostgreSQL database management system. Here are some key aspects of how the PostgreSQL Community works:

Community-driven development: PostgreSQL development is community-driven, with contributors from around the world working together to develop and improve the software. Anyone can contribute to the development of PostgreSQL, and a team of experienced developers reviews all contributions.

Governance: The **PostgreSQL Global Development Group (PGDG)** is the official governing body of the PostgreSQL project. The PGDG manages the overall direction and development of PostgreSQL and makes decisions on critical issues such as major feature development, release management, and community outreach.

Mailing lists and forums: The PostgreSQL community has several mailing lists and forums where developers and users can discuss various PostgreSQL development and usage aspects. These forums provide a platform for discussion, feedback, and collaboration.

Code review and testing: All contributions to PostgreSQL are subject to code review and testing. This ensures that the code is of high quality and meets the community's standards and requirements.

Release management: PostgreSQL releases are managed by the PGDG, with new releases typically occurring every year or two. Each release undergoes an extensive testing and is thoroughly reviewed before being released to the public.

User groups and conferences: The PostgreSQL community has a strong presence in the open source and database communities, with many user groups and conferences held around the world. These events provide opportunities for users and developers to network, learn, and share their knowledge and experiences.

Overall, the PostgreSQL community is a vibrant and collaborative community that values openness, inclusivity, and high-quality software development.

Earning of PostgreSQL community

The PostgreSQL Community is a non-profit organization and does not earn revenue directly. However, the PostgreSQL project receives funding and support from various sources to help sustain and grow the project. Here are some ways that the PostgreSQL Community earns support:

Donations: The PostgreSQL Global Development Group accepts donations from individuals and companies to help fund the development of PostgreSQL. Donations can be made through the PostgreSQL website or through third-party platforms such as Patreon.

Sponsorship: The PostgreSQL Community receives sponsorship from various companies that use and support PostgreSQL. Sponsorship can be in the form of financial support, in-kind contributions, or contributions of developer time.

Consulting services: Many companies offer consulting services for PostgreSQL, providing support, training, and development services to software users. Some of these companies also contribute to the PostgreSQL project through sponsorship or developer time.

Commercial support: Some companies offer commercial support for PostgreSQL, providing support and services to enterprise software users. These companies typically charge for their services but also contribute back to the PostgreSQL project through sponsorship or developer time.

Overall, the PostgreSQL Community relies on the support of its users and supporters to sustain and grow the project. The community is committed to keeping PostgreSQL open source and freely available to all, while also ensuring that the software is of high quality and meets the needs of its users.

Different ways to contribute

There are multiple different ways to contribute to the PostgreSQL Community. If you are interested in being part of this vibrant community, you are plenty of options. In the following paragraphs, we will review the main options you have.

Code contributor

Becoming a code contributor might be among the most engaging and rewarding experiences, but you need to consider this also is one of the contribution options that demand a high knowledge and expertise. *(Reference: PostgreSQL Developer FAQ)*

Before starting to work on a code contribution, you need to be highly familiar with PostgreSQL, know how it works, and its design goals. Fortunately, the code base is available for everyone at the PostgreSQL git repository, you can start your path to becoming a code contributor by getting and studying it.

This is obvious, but being a code contributor means you will participate in the community development processes. The PostgreSQL project has its current status thanks to the highly collaborative development processes. Hence getting involved in the community discussion about bugs, fixes, and enhancements is important.

Another important aspect is to have a clear goal and choose the opportunity area to focus on. The PostgreSQL code is large, robust, and complex, the best option is to define a specific feature or module for which you have identified an opportunity for improvement.

Getting familiar with the code and its design, participating in community discussions, and focusing on a specific module or area will lead us in the correct direction. Once we start working with the code, the next would be the stages:

- **Submit a patch:** Once your code, or patch, is ready, you must submit it to the pgsql-hackers mailing list for review. It needs to be well-documented and follow the code conventions.
- **Actively participate in the review process:** The community may share feedback and suggestion about the patch, it is important to engage and work collaboratively to refine it and ensure it aligns with the general project goals.
- **Get your patch accepted:** After refining and reviewing, the patch will be approved and accepted into the codebase.

Contributing to the codebase will allow you to work and share knowledge with true experts from around the globe. Following the collaborative processes, you can help to make PostgreSQL even better and more powerful.

Bug reporter

Another option to contribute is reporting the bugs we might hit when using, testing, installing, upgrading, and the like, our PostgreSQL databases. Reporting bugs will enable the developers to find fixes to solve them and prevent affecting other users.

To report a bug, there are a few guidelines we can take for a more effective process:

- **Verify if the bug has not already been identified and reported:** The PostgreSQL Community has a TODO list with the already identified bugs; before reporting any, we must verify it. *(Reference: PostgreSQL Bugs and Features)*
- **Create a bug report:** The PostgreSQL Community already has a bug form we can use to report any new bug. It is important to follow the guidelines for bug reports and include all the required information. This will help the developers to reproduce the bug and find the source.
- **Collaborate with the developers:** Following the collaborative perspective of the PostgreSQL Community, you may receive feedback and requests for extra input, it is essential to keep involved in the process so the fix can be addressed as quickly and efficiently as possible.

Identifying bugs is a fundamental way to collaborate and keep PostgreSQL stable and reliable for everyone. Remember, wrong, unclear, or missing documentation is also a bug.

Participate in the mailing lists

The PostgreSQL Community uses different and multiple mailing lists for various purposes. You can collaborate with the community by joining the discussions and sharing knowledge and experience. *(Reference: PostgreSQL Mailing Lists)*

The following are the main aspects you can follow when thinking about participating in the mailing lists:

- **Choose a mailing list:** As said before, there are multiple mail lists for a variety of topics, so you can start by selecting one or two in which you have a special interest or expertise and subscribe to it.
- **Read the mailing list archive:** Before start posting new messages to the mailing list is a good idea to go through the archives; this way, you can get a good sense of the tone and topics for the discussions. Also, it will help to prevent you from posting about already discussed and closed topics.

- **Follow the community guidelines:** The community has adapted some standards to keep the conversation going respectfully and consider others' time and contributions. You need to follow the same.
- **Participate in the discussions:** Contribute to the discussion by sharing your perspective and experience and asking questions. Also, if you know how to answer some of the questions, please do; this will help build your reputation and establish you as a valuable contributor.

Improving or creating new documentation

Getting involved with the Community by helping with the PostgreSQL documentation is also a great option. The is no doubt the official documentation is where everybody looks for references and details while working with our loved PostgreSQL, from anyone taking their first steps into PostgreSQL to experts who need to refer to a particular feature.

Contributing to the project documentation will require you to get familiar with the current documentation, so you know the style, the topics, and the guidelines. Like the previous contribution options, you will need to define an area or module to focus on and get involved with the collaborative work. The following are the main aspects to consider when contributing to the documentation:

- **Get involved with the community:** There is a special mailing list for the PostgreSQL documentation: **pgsql-docs**. You will get the community documentation style guide by joining it and participating. Also, you can identify and confirm an actual documentation opportunity area.
- **Make your contribution:** Once you have defined the area, feature, or module, you can work on your contribution. This could be reporting documentation bugs, fixing them, adding content, or writing new documentation from scratch.
- **Collaborate with the community:** As in previous options, collaborating with the document may require you to share efforts with more community members while receiving feedback or questions to refine your document. Please engage and keep the collaborative spirit.
- **Be persistent:** Documentation is a permanent ongoing effort, invariably, there will be room for improvements. Being persistent and making contributions over time will help to keep PostgreSQL documentation accurate, up-to-date, and accessible for everyone.

Participating in events

The PostgreSQL Community includes many members, organizations, and enterprises. Being as active and vibrant as always, these groups usually organize different events around the globe. These events are a truly awesome way to expand PostgreSQL knowledge and share the experience and enthusiasm for this technology with everyone.

Participating in such events is another fantastic option to contribute to the community. You might consider the following options:

- **Attend a PostgreSQL event:** This is the most straightforward way to participate in an event. It provides an excellent option to network with other community members, learn about new features and best practices, and get involved with the community's ongoing efforts.
- **Speak at a PostgreSQL event:** If you have experience and knowledge working with PostgreSQL, consider submitting a talk proposal to the event. Giving a talk at an event is a terrific way to build your presence as an expert in the community and helps to share knowledge and insights with other members.
- **Volunteer at a PostgreSQL event:** Many events, in part, rely on altruistic volunteer efforts. The volunteers can help with everything, from event registration to planning or technical support. Collaborating this way can ensure the event's success. Also, it is a good way to get involved with the community and engage with the members.
- **Organizing a PostgreSQL event:** If you have some experience in this regard, you can organize an event in your local area. This can be from a small group meeting to a full-blown conference. Even when this option involves a lot of work, it will help to build a network, establish yourself as a leader in the community, and contribute to the ongoing success of PostgreSQL.

The event options are vast, listing some of the most important: *(Reference: PostgreSQL Events)*

- **PostgreSQL Conference:** This is the official PostgreSQL conference organized by the PostgreSQL Global Development Group. It is held annually in different locations worldwide and includes talks and workshops covering various PostgreSQL-related topics.
- **PGConf:** PGConf is a series of PostgreSQL conferences held in different locations around the world. The PostgreSQL community organizes these events and typically includes talks, workshops, and networking opportunities for PostgreSQL users, developers, and contributors.

- **PostgresOpen:** PostgresOpen is an annual conference for PostgreSQL users and developers held in different locations around the world. The conference includes talks, tutorials, and workshops covering various PostgreSQL-related topics.
- **FOSDEM:** FOSDEM is a free and open-source software conference held annually in Brussels, Belgium. While not exclusively focused on PostgreSQL, the conference typically includes talks and workshops related to PostgreSQL and opportunities to meet and network with other PostgreSQL community members.

Supporting the community

All the previous options to contribute to the PostgreSQL Community are related to your experience with PostgreSQL or your availability to participate directly in the different aspects the community works on. However, there is still at least another option to participate with the community, which does not require your having experience or time to get involved with specific work. This is being a financial sponsor.

If you can support the community financially, you can do it at least in the two following ways:

- **Donate directly to the PostgreSQL project:** PostgreSQL is a free and open-source project, but it relies on donations to support ongoing development and maintenance. You can donate directly to the PostgreSQL project through the PostgreSQL Foundation's website. *(Reference: PostgreSQL Donate)*
- **Sponsor PostgreSQL events:** Many PostgreSQL events rely on the support of sponsors to cover costs and provide additional resources for attendees. By sponsoring a PostgreSQL event, you can help ensure that the event is a success and contribute to the ongoing development of PostgreSQL.

Conclusion

With this, we came to the end of this chapter and the end of this book. To conclude, we have covered not only the concepts of PostgreSQL but also how the PostgreSQL Community is managed and how one can contribute to the community.

The PostgreSQL Community is one of the most important parts of the PostgreSQL software. The software will survive as long as Community is there; the vice versa may/may not be true. It is WE, as a user, who can make the community stronger by contributing not only in the form of code but also in other ways discussed in this chapter.

Bibliography

- Tom Lane: **https://en.wikipedia.org/wiki/Tom_Lane_(computer_scientist))**
- PostgreSQL Developer FAQ: **https://wiki.postgresql.org/wiki/Developer_FAQ#Getting_Involved**
- PostgreSQL Bugs and Features: **https://wiki.postgresql.org/wiki/FAQ#How_do_I_find_out_about_known_bugs_or_missing_features.3F**
- PostgreSQL Mailing Lists: **https://www.postgresql.org/list/**
- PostgreSQL Events: **https://www.postgresql.org/about/events/**
- PostgreSQL Donate: **https://www.postgresql.org/about/donate/**

Index

A

aggregate functions 209
Amazon Web Services (AWS) 46
archiver 79
archive recovery 139, 140
Atomicity 92
Atomicity, Consistency, Isolation, and Durability (ACID) 16, 92
authentication
  about 152
  address 155, 156
  database 154, 155
  method 156
  [Options] 156, 157
  pg_hba.conf 152, 153
  user 155
authentication methods
  about 157
  pg_ident.conf 158, 159
authorization
  about 161
  object ownership 163
  object privileges 164, 165
  role attributes 161, 162
autovacuum 79, 94

B

background processes
  about 77
  archiver 79
  autovacuum 79
  background writer 78
  checkpointer 78
  logger 79
  postmaster 77
  stats collector 79
  WAL receiver 81
  WAL sender 80
  WAL writer 80
  writer 78
background writer 78
backup
  about 108
  logical backup 112
  pg_basebackup 109
backup and restore tools
  about 118
  Barman 125, 126
  pgBackRest 119-125
  pg_probackup 127-133
Barman 125, 126

Binary installation
  about 33, 34
  package list, updating 35
  repository configuration, creating 34
  repository signing key, importing 35
Block Range INdexes (BRIN) 237
Brin index 237
b-tree index 236

C

cascading 142, 143
checkpointer 78
Classless Inter-Domain Routing (CIDR) 155
CLOG buffer 75
Command-Line Interface (CLI) 174
Commit Log (CLOG) 75
Commit LOG Files (CLOG Files) 86
Consistency 92
constraints
  used, for enforcing data integrity 197, 199
CREATE DATABASE command 67, 68
createdb program 68
cron 170
custom data type 230

D

data
  manipulating, with DML Queries 201
database
  about 65, 66
  CREATE DATABASE command 67, 68
  createdb program 68
  pgAdmin Wizard, using 69-72
Database Administrators (DBAs) 179, 233
Data Definition Language
      (DDL) 147, 185, 192
data files 84
data integrity
  enforcing, with constraints 197-200
data types 192, 193
DB cluster 186
DB Objects
  data types 192, 193
  managing, with DDL commands 192
  table 193
DDL commands
  used, for managing DB Objects 192
debugging
  with log files 256
  with Operating System tools
      and commands 268
  with PostgreSQL tools and commands 261
Defense Advanced Research
      Projects Agency (DARPA) 6
delayed replica 144-146
df command 270
Digital Ocean (DO) 46
Disaster Recovery (DR) models 135
distributed architecture
  with PostgreSQL 17
distributed architecture, PostgreSQL
  data partitioning 18
  horizontal scaling 17
  replication 18
DML Queries
  about 102, 147, 185, 201
  aggregate functions 209, 210
  data, deleting 203
  data, inserting 201, 202
  data, selecting 203
  data, updating 202
  full outer join 208
  inner join 204, 205
  joins, using in data retrieval 204
  used, for manipulating data 201
Docker 47
Docker Hub
  reference link 47
Domain Name System (DNS) 156
Durability 92

E

event trigger 227
executor 105
explain plan 248, 250
expressions 237
extension
  about 168
  pg_cron 169, 170
  pg_repack 174-177
  pg_stat_statements 171, 172
extension tools
  about 177
  pgbadger tool 177-179
  pgbench tool 179-181
  pgbouncer tool 181-183

F

Free and Open-source Software (FOSS) 7
free command 269
Free Software Foundation (FSF) 3, 5
full outer join 208
function
  about 212, 213
  managing 212
function execution
  example 215
  syntax 214

G

General Public License (GPL) 3
Generic Security Service Application Program Interface (GSSAPI) 154
Geographic Information Systems (GIS) 17
Gin index 237
GiST index 237
Google Cloud Platform (GCP) 46
groups 60-63

H

hash index 237
High-Availability (HA) 135
hostnogssenc 154
hypervisor 43

I

indexes
  about 234, 235, 238
  reindex 235, 236
index types
  about 236
  Brin index 237
  Btree index 236
  Gin index 237
  GiST index 237
  hash index 237
  SP-GiST index 237
inner join
  about 204, 205
  left outer join 206
  right outer join 207
Isolation 92

J

journalctl 272

L

left outer join 206
libre software 2
Linux kernel 4
log files
  about 86
  debugging with 256
  log_autovacuum_min_duration 260
  log_connections 260
  log_destination 257
  log_directory 257
  log_disconnections 260
  log_filename 258
  logging 259
  logging_collector 256
  log_line_prefix 259

log_lock_waits 260
log_min_duration_statement 260
log_rotation_age 258
log_rotation_size 258
log_truncate_on_rotation 258
parameter summary 261
storing 256
logger 79
logical backup
about 112
cons 115
examples 118
pg_dump 112
pg_dumpall 112, 114
pros 115
logical replication
about 146
architecture 147
limitations 150
publication 147
publisher node 148, 149
subscription 148
subscription node 148, 149
use cases 149
log_line_prefix 259
Log Sequence Number (LSN) 85

M

manual VACUUM
about 95
options 95
pg_repack command 96
phases 96
memory architecture
about 74
maintenance work memory 77
process memory 76
shared buffer 75
shared memory 74
WAL buffer 75
work memory 76
Microsoft Azure 46
MINIX 4
Multiversion Concurrency
Control (MVCC) 8, 16, 75, 93, 174

N

National Science Foundation (NSF) 6
Netscape browser 4

O

open-source
about 2
concept 3-5
free software 2, 3
overview 5, 6
Operating System tools and commands
debugging with 268
log and events 272
pgrep commands 271
processes-oriented tools 270
ps commands 271
service and system-wide tools 268

P

parser
about 103
join tree 104
others 104
qualification 104
Range Table Entry (RTE) 104
result relation 104
target list 104
type 104
PersistentVolumeClaim (PVC) 48
PersistentVolume (PV) 48
pgAdmin Wizard
using 69-72
pgBackRest 120-125
pgbadger tool 177-179
pgbench tool 180, 181
pgbouncer tool 181-183

pg_cron 169
pg_dump 112
pg_dumpall 112, 114
pg_hba.conf
  about 152, 153
  host 153
  hostgssenc 154
  hostnogssenc 154
  hostnossl 154
  hostssl 154
  local 153
pg_ident.conf
  about 158, 159
  examples 159-161
pg_probackup 127-133
pg_reorg 176
pg_repack 174-176
pg_repack command 96
pg_restore 117
pg_stat_statements 171, 172
phantom read 99
phenomena types
  dirty read 97
  non-repetable read 98
physical backup
  about 109
  cons 111
  pros 111
physical files
  Commit LOG Files (CLOG Files) 86
  log files 86
  stat files 86
  temporary files 85
  WAL archive files 86, 87
  WAL files 84, 85
physical file structure
  data files 84
  defining 81-83
physical replication
  about 136-138
  archive recovery 139, 140
  cascading 142, 143
  configuration 146
  delayed replica 144-146
  hot standby 138, 139
  streaming replication 141, 142
planner 104
Point in Time Recovery (PITR) 17, 111
PostGIS extension 17
Postgres 1, 61
Postgres95 7
POSTGRES project 6
PostgreSQL
  2ndQuadrant 15
  about 7, 42
  advantages 16
  cloud 45, 46
  containers 44, 45
  database 31
  data directory, validating 30
  distributed architecture 17
  enhancing 14, 15
  EnterpriseDB 15
  extension, adding 173
  history 6
  initializing 28
  installing 36-38
  market impact 12, 13
  on cloud 53-57
  on Docker 47, 48
  on Kubernetes 48, 49, 52, 53
  on modern systems 46
  on-premise 42, 43
  Pivotal 15
  Postgres95 7
  postgres process, verifying 31, 32
  POSTGRES project 6
  statefulset protocol, using 18
  using 14
  virtualization 43, 44
PostgreSQL community
  about 276

bug reporter 280
code contribution 279
core members 276, 277
documentation, creating 281
documentation, improving 281
earning 278
event participating 282, 283
mailing lists, participating 280, 281
reference link 86
supporting 283
working pattern 277, 278
postgresql.conf parameters, best practices
about 250
autovacuum 251
effective_cache_size 252
maintenance_work_mem 252
max_connections 252
shared_buffers 251
work_mem 251
PostgreSQL members 276
PostgreSQL release cycle 10, 11
PostgreSQL stats
on single image 10
PostgreSQL tools and commands
ANALYZE operations, executing 266
database and object size, checking 262
database connections, checking 263
debugging with 261
Instruct PostgreSQL 266
PostgreSQL version, checking 262
replication, managing 267
slow queries, checking 264
statistics, checking 264, 265
VACUUM operations, executing 266
PostgreSQL versions
key features 8
versions 6.0 - 8.0 8
versions 8.1 - 9.6 8, 9
versions 10 - 14 9, 10
postmaster 77
predefined roles 63
procedure
about 216, 217
managing 212
procedure execution
about 219, 220
syntax 217-219
processes-oriented tools
about 270
htop tool 270
top tool 270
process memory
about 76
temporary buffer 76
psql command 115, 116
public schema
about 188
SEARCH_PATH attribute 189-191

Q

queries
terminating 266
query processing
about 102, 103
executor 105
parser 103
planner 104
rewriter 104

R

reindex 235, 236
Relational Database Management System (RDBMS) 73, 97, 186, 192
Relational Database Service (RDS) 54
replication & high availability
advance features 17
restore
about 115
pg_restore 117
psql command 115, 116

rewriter 104
right outer join 207
role attributes 161, 162
roles 60-63
rules
  managing 227, 228
  versus trigger 230

S

schemas
  about 188
  database 188
  DB cluster 186, 187
  managing 186
  public schema 188
  tablespaces 188
  users/roles 188
SEARCH_PATH attribute 189, 190
serialization anomaly 99, 100
service and system-wide tools
  about 268
  df command 270
  free command 269
  systemctl 268, 269
shared buffer 75
shared memory 74
source code installation
  about 22
  downloading 23, 24
  pre-requisites 23
  version 22, 23
source code installation process
  about 25, 26
  data directory, creating 28
  directory structure, verifying 27
  postgres user, adding 27
SP-GiST index 237
statefulset protocol
  with PostgreSQL 18
stat files 86
statistics
  about 238-240
  common values list 246, 247
  functional dependencies 243, 244
  in pg_statistics 240-242
  in pg_statistics_ext_data 242, 243
  number of distinct values counts 244, 245
  system catalogs 238
  types 238
stats collector 79
streaming replication 141
subscription 148
superuser 161
systemctl 268, 269

T

table
  about 193
  altering 194
  creating 193
  dropping 194
  sequences 195, 196
  truncating 194
  viewing 195
tablespace 64, 65
temporary buffer 76
temporary files 85
transaction isolation levels
  about 97
  phenomena 97
  phenomena isolation levels 100
  read committed 100
  read uncommitted 100
  repeatable read 101
  serializable 102
Transmission Control Protocol/
    Internet Protocol (TCP/IP) 153
trigger
  versus rules 230

trigger function 222-226
triggers
  managing 222

U

UNiplexed Information Computing System (UNIX) 4
users 60-63
user sessions
  terminating 266

V

vacuum
  about 94
  autovacuum 94
  manual VACUUM 95
  transaction ID wraparound failures, preventing 97
  VACUUM FULL 95
VACUUM FULL 95
Virtual Machines (VM) 43

W

WAL archive files 86, 87
WAL buffer 75
WAL files 84, 85
WAL receiver 81, 141
WAL sender 80, 141
WAL writer 80
work memory 76
Write-Ahead Logging (WAL) 8, 16, 75, 80, 136
writer 78

PostgreSQL for Jobseekers

DESCRIPTION

PostgreSQL is a powerful open-source relational database management system (RDBMS) that is widely used in the industry. If you are seeking to acquire knowledge about PostgreSQL, this book is for you.

This comprehensive book provides you with a solid foundation in working with PostgreSQL, a popular open-source database management system. It covers a broad spectrum of topics, allowing you to successfully install and configure PostgreSQL across various platforms and methods. By delving into the internal components that constitute a PostgreSQL service and their interplay, you will gain a deep understanding of how these elements collaborate to deliver a robust and dependable solution. From comprehending the process model and shared memory to mastering query execution and optimization, you will acquire comprehensive knowledge of PostgreSQL's internal workings. Furthermore, the book explores essential tasks performed by a database administrator (DBA), including backup and restore operations, security measures, performance tuning, and troubleshooting techniques. Lastly, it explores widely used extensions and compatible tools that can enhance the functionality of PostgreSQL.

Upon completing this book, you will have developed a comprehensive understanding of the internal components that comprise a PostgreSQL service and their collaborative dynamics, resulting in a reliable and robust solution.

KEY FEATURES

- Acquire in-depth knowledge of PostgreSQL's key capabilities and gain a comprehensive understanding of its inner workings.
- Discover the art of extending PostgreSQL's core features and effectively troubleshooting any challenges that may arise.
- Explore the vibrant community and open-source ecosystem that forms the foundation of PostgreSQL's development and innovation.

WHAT YOU WILL LEARN

- Gain proficiency in installing and preparing PostgreSQL for various methods and platforms.
- Develop a solid understanding of the internal components of a PostgreSQL service and their collaborative dynamics to deliver a comprehensive solution.
- Acquire knowledge about essential tasks performed by PostgreSQL DBAs, including backup/restore operations, security measures, tuning, and troubleshooting.
- Explore popular extensions and compatible tools that can expand and enhance the capabilities of PostgreSQL.
- Discover the PostgreSQL Community and learn how to actively contribute to the project's development and growth.

WHO THIS BOOK IS FOR

This book is highly recommended for Entry Level Database Administrators, as it provides a suitable starting point for their journey. It assumes some prior knowledge of Database Management Systems (DBMS) to ensure a smooth learning experience. Additionally, senior or experienced developers will find value in this book, particularly in gaining insights into the latest features incorporated in the most recent version of the DB, enhancing their understanding and proficiency in its use.

BPB PUBLICATIONS
www.bpbonline.com

ISBN 978-93-5551-400-4