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.