Chapter 13 - Performance Tuning

Introduction

Any Relational Database Management System (RDBMS) aims to resolve user queries as quickly as possible, and PostgreSQL is no exception. Therefore, maintaining good performance in PostgreSQL is one of the main goals for Database Administrators (DBAs).

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

Structure

During this chapter, we will study the following sections:

- Indexes
- Statistics
- Execution Plan (Explain Plan)
- Best practices for `postgresql.conf` parameters

Objectives

By completing this chapter, you will be familiar with the basic components for tuning PostgreSQL and understand their relationship and impact. You will also learn to apply standard best practices when installing or tuning a PostgreSQL system to achieve the best performance.

Indexes

Have you ever noticed that book indexes help locate chapters by mentioning the page number? This allows you to jump directly to that page instead of sequentially going through each chapter, saving time. Similarly, indexes in PostgreSQL create metadata, like page numbers, that help directly access the requested data instead of scanning the entire table sequentially.

However, this benefit comes with a cost since adding an index requires creating or modifying it as data is inserted, updated, or deleted in the table. Therefore, indexes should be created strategically after proper analysis. If the table is used more for read operations than for write operations, indexes can be beneficial. Otherwise, they may become an additional overhead since they need to be updated with every write operation.

```
CREATE [UNIQUE] INDEX [CONCURRENTLY] [IF NOT EXISTS] name ON
[ONLY] table_name [USING method]
  (column | (expression) [COLLATE collation] [opclass ...])
  [INCLUDE (column, ...)]
  [NULLS [NOT] DISTINCT]
  [WITH (storage_parameter = value, ...)]
  [TABLESPACE tablespace_name]
  [WHERE predicate]
```

Syntax according to PostgreSQL documentation: Creating and dropping an index (`CREATE INDEX` and `DROP INDEX`) is not an online activity by default. The table is locked for any modifications during the operation. However, read operations can still be performed while an index is being created or modified if the `CONCURRENTLY` keyword is used. This allows indexes to be modified concurrently without locking the table.

Indexes can also be created on multiple columns, known as multi-column indexes. When doing so, it is important to ensure that the order of the columns in the index matches the order used in the query's `WHERE` clause.

Reindexing (REINDEX)

Reindexing an index is a PostgreSQL maintenance activity that helps rebuild corrupted, bloated, or invalid indexes concurrently. An index can also be rebuilt when its storage location is changed.

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

Syntax according to PostgreSQL documentation: Where the options can be:

- `CONCURRENTLY [boolean]`: Allows reindexing to be performed concurrently.
- `TABLESPACE new_tablespace`: Allows reindexing in a new tablespace.
- `VERBOSE [boolean]`: Displays detailed logs during reindexing.

When Primary Key (`PRIMARY KEY`) or Unique Key (`UNIQUE`) constraints are created, the system automatically generates indexes on those columns, as they are commonly used for data retrieval. Generally, indexes should be created on columns used in the `WHERE` clause. Although this is not a strict rule, it is good practice to create indexes on columns used as foreign keys (`FOREIGN KEY`), as they are often involved in `JOIN` conditions for data retrieval.

In small databases, the default indexes created on Primary or Unique Keys may be sufficient at first. However, as data grows, queries may become slower. The `EXPLAIN PLAN` command can be used to check whether queries are utilizing indexes. This command will be discussed in more detail later in this chapter.

Indexes are a key component for improving query performance, especially for queries involving large amounts of data or `SELECT` queries that require optimized execution times.

They can also improve the performance of `INSERT`, `UPDATE`, and `DELETE` queries when a `WHERE` condition is used. Running the `ANALYZE` command is recommended to keep index statistics up to date, ensuring they contain recent metadata that helps enhance query performance.

Types of Indexes

Each query is different, and the default `BTREE` index may not always be the best option. To address this, PostgreSQL provides various types of indexes that can improve the performance of different types of queries. To specify an index type other than `BTREE`, the `USING` keyword is used in the `CREATE INDEX` statement.

B-Tree (BTREE) Index By default, PostgreSQL creates a `BTREE` index if no other type is specified. This index sorts the data sequentially and works well when the equality operator is used in the `WHERE` clause. It is also efficient with the following operators:

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

B-Tree indexes can also be used to retrieve sorted data. Although this is not always faster than a simple scan and sort operation, it is often useful.

Hash Index This type of index generates a 32-bit hash code on the column where the index is created. It is most efficient when using the equality operator in the query's `WHERE` clause.

GiST and SP-GiST Index These indexes are useful when working with two-dimensional geometric data types.

GIN Index This index is used when dealing with one-dimensional data types like `ARRAYS`.

BRIN Index BRIN (Block Range INdex) indexes store summaries of values stored in consecutive physical block ranges of a table. They are most effective for columns whose values are well correlated with the physical order of the table rows.
