A step-by-step guide to SQL performance tuning

Ansam Yousry · Follow
Published in ILLUMINATION · 13 min read · Jan 27

SQL Performance Tuning, image created by author, © the author assumes responsibility for the provenance and copyright.

Unlock the secrets of lightning-fast SQL performance with our in-depth guide to SQL performance tuning. From optimizing query structure to using indexes, partitioning, and caching, we'll show you insider tricks and tips to make your database run like a well-oiled machine. Learn how to handle large amounts of data and high traffic with ease, and keep your business running smoothly. Don't let slow SQL queries hold you back — read our article now and take your database performance to the next level

Table Of Contents:

Keys of SQL performance tuning:

SQL performance tuning involves analyzing and optimizing various components of a SQL query in order to make it execute as quickly and efficiently as possible. Some of the key areas that can be optimized include:

- **Query structure:** This includes things like using proper joins, filtering data early in the query, and avoiding unnecessary subqueries.
- **Indexing:** Indexes can be created on columns that are frequently searched or sorted in order to speed up those operations.
- **Database configuration:** This includes things like adjusting memory and buffer pool settings, adjusting the settings of the query optimizer, and configuring the database to use the appropriate storage engine.
- **Partitioning:** Partitioning large tables can improve query performance by allowing the database to only scan the partitions that are relevant to a given query.
- **Caching:** Caching query results can improve performance by reducing the need to repeatedly run the same query.
- **Data distribution**: Data should be distributed evenly across the storage devices to reduce contention and increase performance.

To perform SQL performance tuning, the database administrator or developer needs to monitor the performance of the database using tools such as the query optimizer and performance monitors, analyze the performance bottlenecks, and make adjustments accordingly. It's an ongoing process that requires continuous monitoring and adjustment as the data and usage patterns of the database change.

. . .

Query structure:

The structure of a SQL query can have a significant impact on its performance. Properly structuring a query can help the database server more efficiently retrieve the requested data. Here are a few examples of how to query structure can be used to improve performance:

1. **Using proper joins**: Joining tables in a query can be a powerful way to retrieve related data, but it can also be a performance bottleneck. When joining tables, it's important to use the appropriate type of join (e.g. inner join, left join, etc.) and to join on indexed columns to improve performance.

For example, the following query is more efficient than the one that uses a subquery to retrieve data from multiple tables:

```
SELECT *
FROM orders
JOIN customers ON orders.customer_id = customers.customer_id
JOIN products ON orders.product_id = products.product_id;
```

2. **Filtering data early in the query**: The order of the clauses in a query can also impact performance. It's generally more efficient to filter data as early as possible in the query so that the database server doesn't need to retrieve and process unnecessary data.

For example, the following query is more efficient than the one that retrieves all data and filters it later:

```
SELECT *
FROM orders
WHERE order_date >= '2022-01-01'
AND order_date < '2022-02-01';
```

3. **Avoiding unnecessary subqueries**: Subqueries can be useful for retrieving related data, but they can also be a performance bottleneck. Avoid using subqueries when it is possible to retrieve the same data using a join or by moving the subquery to a separate query.

For example, the following query is more efficient than the one that uses a subquery to retrieve data from multiple tables:

```
SELECT *
FROM orders
WHERE customer_id IN (SELECT customer_id FROM customers WHERE country = 'USA');
```

It's worth noting that, even if you have a well-structured query, other factors like indexing, database configuration, and hardware resources can also play important role in the performance of the query. It's important to monitor the performance of the query and make adjustments as needed.

. . .

Indexing:

Indexing is a technique that can be used to improve the performance of SQL queries by allowing the database server to quickly locate the requested data. Indexes work by creating a separate data structure that stores a mapping of the indexed column's values to the corresponding rows in the table. Here are a few examples of how indexing can be used to improve performance:

1. **Creating indexes on frequently searched columns**: When a query includes a WHERE clause that filters data based on a specific column, an index on that column can be used to quickly locate the relevant rows. For example, creating an index on the "customer_id" column in the "orders" table can improve the performance of a query like this:

```
SELECT * FROM orders WHERE customer_id = 123;
```

2. **Creating indexes on frequently sorted columns**: When a query includes an ORDER BY clause that sorts the data based on a specific column, an index on that column can be used to quickly sort the data. For example, creating an index on the "order_date" column in the "orders" table can improve the performance of a query like this:

```
SELECT * FROM orders ORDER BY order_date;
```

3. **Creating composite indexes:** A composite index is an index that is based on multiple columns. Creating a composite index on columns that are frequently used together in WHERE clauses or ORDER BY clauses can improve performance. For example, creating a composite index on the “customer_id” and “order_date” columns in the “orders” table can improve the performance of a query like this:

```
SELECT * FROM orders WHERE customer_id = 123 ORDER BY order_date;
```

4. **Covering index:** Covering indexes are indexes that include all the columns that are being selected in the query. They can be used to improve performance by reducing the number of disk I/O operations that are required to retrieve the requested data. For example, creating a covering index on the “customer_id”, “order_date” and “product_name” columns in the “orders” table can improve the performance of a query like this:

```
SELECT customer_id, order_date, product_name FROM orders WHERE customer_id = 123;
```

Creating indexes in SQL is typically done using the CREATE INDEX statement. The syntax for this statement varies depending on the specific SQL database management system you are using, but the basic structure is the same. Here are a few examples of how to create indexes in different SQL databases:

1. **MySQL**

```
CREATE INDEX index_name ON table_name (column_name);
```

For example, the following command creates an index named “customer_id_index” on the “customer_id” column in the “customers” table:

```
CREATE INDEX customer_id_index ON customers (customer_id);
```

2. **SQL Server**

```
CREATE INDEX index_name ON table_name (column_name);
```

For example, the following command creates an index named “order_date_index” on the “order_date” column in the “orders” table:

```
CREATE INDEX order_date_index ON orders (order_date);
```

3. **Oracle**

```
CREATE INDEX index_name ON table_name (column_name);
```

For example, the following command creates an index named "product_name_index" on the "product_name" column in the "products" table:

```
CREATE INDEX product_name_index ON products (product_name);
```

4. **PostgreSQL**

```
CREATE INDEX index_name ON table_name (column_name);
```

For example, the following command creates an index named "product_id_index" on the "product_id" column in the "orders" table:

```
CREATE INDEX product_id_index ON orders (product_id);
```

It's worth noting that, each database management systems have its own specific functionalities and options while creating indexes. It's important to consult the documentation of the specific database management system you're using for more information on creating indexes.

. . .

Database configuration:

Database configuration plays an important role in the performance of a SQL database. Properly configuring a database can help to optimize the performance of queries, reduce resource usage, and improve overall system stability. Here are a few examples of how database configuration can be used to improve performance:

1. **Configuring the buffer pool**: The buffer pool is a memory area that is used to cache data pages from the database. Configuring the buffer pool to be large enough to cache frequently accessed data can help to reduce the number of disk I/O operations that are required to retrieve the data. For example, increasing the buffer pool size in MySQL can improve the performance of queries that access frequently accessed tables.
2. **Configuring the query cache**: The query cache is a memory area that is used to cache the results of SELECT statements. Configuring the query cache to be large enough to cache the results of

frequently executed queries can help to reduce the number of times the query needs to be executed. For example, enabling the query cache in MySQL can improve the performance of frequently executed SELECT statements.

3. **Configuring the log files**: Log files are used to store information about the operations that are performed on the database. Configuring the log files to be stored on a separate disk or partition can help to prevent disk I/O bottlenecks that can occur when the log files are stored on the same disk as the data files. For example, configuring the log files to be stored on a separate disk in SQL Server can improve the performance of queries that access large tables.
4. **Configuring the memory**: Properly configuring the memory usage of a database can help to improve the performance of queries. For example, increasing the amount of memory that is allocated to the database server can help to reduce disk I/O operations and improve query performance.
5. **Configuring the connection pool**: Connection pooling is the technique of maintaining a pool of open connections to the database. Configuring the connection pool effectively can help to reduce the time required to establish new connections, thus improving the performance of queries.

It's worth noting that, the best configuration settings for your database will depend on your specific use case and workload. It's important to monitor the performance of your database and make adjustments as necessary to ensure that it is configured optimally.

. . .

Partitioning:

Partitioning is a technique used to divide a large table into smaller, more manageable pieces called partitions. Partitioning can improve the performance of a SQL database by allowing for more efficient data retrieval and manipulation. Here are a few examples of how partitioning can be used to improve performance:

1. **Range partitioning**: In this type of partitioning, data is divided into partitions based on a range of values, such as date or price range. This allows for more efficient data retrieval because the database can quickly locate the partition that contains the data it needs. For example, if you have a table that stores sales data, you could partition the table by date range so that all data for a specific month is stored in the same partition. This would make it faster to retrieve sales data for a specific month.

```
CREATE TABLE sales_data (
    sales_id INT NOT NULL,
    sales_date DATE NOT NULL,
    sales_amount DECIMAL(10,2) NOT NULL,
    product_id INT NOT NULL
)
PARTITION BY RANGE (sales_date) (
    PARTITION p0 VALUES LESS THAN ('2022-01-01'),
    PARTITION p1 VALUES LESS THAN ('2022-06-01'),
    PARTITION p2 VALUES LESS THAN ('2022-12-01'),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

This creates a table named “sales_data” that is partitioned by the “sales_date” column. The table is divided into 4 partitions, one for each quarter of the year. Data for each quarter is stored in a separate partition, making it faster to retrieve data for a specific quarter.

2. **Hash partitioning**: In this type of partitioning, data is divided into partitions based on a hash function. This allows for more efficient data retrieval because the database can quickly locate the partition that contains the data it needs. For example, if you have a table that stores customer data, you could partition the table by customer ID so that all data for a specific customer is stored in the same partition. This would make it faster to retrieve data for a specific customer.

```
CREATE TABLE customer_data (
    customer_id INT NOT NULL,
    customer_name VARCHAR(50) NOT NULL,
    customer_address VARCHAR(255) NOT NULL
)
PARTITION BY HASH (customer_id)
PARTITIONS 5;
```

This creates a table named “customer_data” that is partitioned by the “customer_id” column. The table is divided into 5 partitions, each one determined by a hash function applied to the customer_id. Data for each customer is stored in a specific partition, making it faster to retrieve data for a specific customer.

3. **List partitioning**: In this type of partitioning, data is divided into partitions based on a specific list of values. This allows for more efficient data retrieval because the database can quickly locate the partition that contains the data it needs. For example, if you have a table that stores product data, you could partition the table by product category so that all data for a specific category is stored in the same partition. This would make it faster to retrieve data for a specific product category.

```
CREATE TABLE product_data (
    product_id INT NOT NULL,
    product_name VARCHAR(50) NOT NULL,
    product_price DECIMAL(10,2) NOT NULL,
    product_category VARCHAR(50) NOT NULL
)
PARTITION BY LIST (product_category) (
    PARTITION p_books VALUES IN ('Books'),
    PARTITION p_electronics VALUES IN ('Electronics'),
    PARTITION p_clothes VALUES IN ('Clothes'),
    PARTITION p_others VALUES IN (NULL)
);
```

This creates a table named “product_data” that is partitioned by the “product_category” column. The table is divided into 4 partitions, one for each category of product. Data for each category is stored in a separate partition, making it faster to retrieve data for a specific category.

. . .

Caching:

There are several ways caching can be used to improve the performance of SQL queries. Here are a few examples:

1. **Using the query cache**:

```
SET GLOBAL query_cache_size = 1073741824; -- 1 GB
SET GLOBAL query_cache_type = 1; -- Enable query cache
```

The above SQL statements increase the size of the query cache to 1 GB and enable the query cache. Once enabled, the query cache stores the result of SELECT statements in memory, so that if the same SELECT statement is executed again, the result can be retrieved from the cache instead of being recalculated. This can significantly improve the performance of frequently executed SELECT statements.

2. **Using the InnoDB buffer pool**:

```
SET GLOBAL innodb_buffer_pool_size = 1073741824; -- 1 GB
```

The InnoDB buffer pool is a memory area that caches data and indexes for InnoDB tables. By increasing the size of the buffer pool, you can increase the amount of data and indexes that can be cached in memory. This can improve the performance of SELECT, UPDATE, and DELETE statements that access InnoDB tables.

3. **Using the Memcached:**

```
SELECT * FROM table_name WHERE id = 10;
```

```
SELECT SQL_CACHE * FROM table_name WHERE id = 10;
```

```
SELECT SQL_NO_CACHE * FROM table_name WHERE id = 10;
```

The above SQL statement retrieves data from the table “table_name” where the “id” column is equal to 10. The SELECT SQL_CACHE statement retrieves the data from the cache if it is already present, otherwise, it retrieves the data from the table and stores it in the cache. The SELECT SQL_NO_CACHE statement retrieves the data directly from the table, bypassing the cache.

It’s worth noting that, caching can be applied to different types of databases, but it may have different implementations and syntax. It’s important to consult the documentation of the specific database management system you’re using for more information on caching.

. . .

Data distribution:

Data distribution refers to the process of distributing data across multiple servers or storage devices to improve performance. There are a few ways data distribution can be used to enhance performance in SQL:

1. **Horizontal partitioning:**

```
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    order_date DATE,
    total_cost DECIMAL(10,2)
)
PARTITION BY RANGE (order_date) (
    PARTITION p0 VALUES LESS THAN ('2022-01-01'),
    PARTITION p1 VALUES LESS THAN ('2022-06-01'),
    PARTITION p2 VALUES LESS THAN ('2022-12-01'),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

In the above example, the table "orders" is partitioned by range on the "order_date" column. This means that the table is divided into multiple partitions, each containing a range of "order_date" values. This allows for more efficient queries on large data sets by only accessing the relevant partition rather than scanning the entire table.

2. **Vertical partitioning:**

```
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    order_date DATE
);

CREATE TABLE order_items (
    order_id INT,
    item_id INT,
    quantity INT,
    price DECIMAL(10,2)
);
```

In the above example, the table "orders" is separated into two tables: "orders" and "order_items". The "orders" table contains information about the order (order_id, customer_id, order_date) and the "order_items" table contains information about the items in the order (order_id, item_id, quantity, price). This allows for more efficient queries on large data sets by only accessing the relevant table rather than scanning the entire table.

3. **Sharding:**

```
CREATE TABLE orders_shard_1 (
    order_id INT,
    customer_id INT,
    order_date DATE,
    total_cost DECIMAL(10,2)
);

CREATE TABLE orders_shard_2 (
    order_id INT,
    customer_id INT,
    order_date DATE,
    total_cost DECIMAL(10,2)
);
```

In the above example, the table “orders” is split into multiple tables, each containing a subset of the data. Each table is called a “shard”. This allows for more efficient queries on large data sets by only accessing the relevant shard rather than scanning the entire table. It also allows for better scalability by distributing data across multiple servers.

It’s worth noting that, data distribution can be applied to different types of databases, but it may have different implementations and syntax. It’s important to consult the documentation of the specific database management system you’re using for more information on data distribution.

You can read more about performance tuning here or here

conclusion:

SQL performance tuning is the process of optimizing the performance of SQL queries and database operations. There are several techniques that can be used to improve performance, such as optimizing query structure, using indexes, configuring the database, partitioning, caching, and data distribution. By using these techniques, you can improve the performance of your SQL queries and ensure that your database can handle large amounts of data and handle high levels of traffic. However, it’s worth noting that the specific techniques and their implementations may vary depending on the database management system you’re using. It’s important to consult the documentation of your specific system for more information on how to optimize performance.

I hope you enjoyed reading this and finding it informative, feel free to add your comments, thoughts, or feedback, and don’t forget to get in touch on LinkedIn or follow my medium account to keep updated.

Did you like this article? Don’t forget to hit the Follow button and you might also like:

SQL Window Functions With Examples

In SQL, a window function (also called an “over clause”) allows you to perform calculations on a set of rows that are…

medium.com

Apache Spark vs Hadoop

Main differences between Hadoop and Spark

blog.devgenius.io

Nano Degree Program for Data Engineering — Does it worth getting?

I enrolled in the Nano degree for data engineering and in this article, I will share with you my opinion and more…

blog.devgenius.io