Performance Optimization

Query Optimization Fundamentals

Query optimization is a crucial aspect of database management that ensures efficient execution of SQL queries by improving speed, reducing resource consumption, and ultimately enhancing the overall performance of the system.

1. Understanding the Query Execution Process

Before optimization happens, a query goes through several steps:

1. **Query Parsing**: SQL queries are parsed to check for syntax errors and generate a parse tree.
2. **Rewriting**: The query may be rewritten by the optimizer to simplify or normalize it.
3. **Query Planning**: PostgreSQL's planner/optimizer generated multiple execution plans, estimating costs based on factors like indexes, join methods and data distribution.
4. **Statistics**: Uses table and index statistics (rows count, data distribution) collected by the `ANALYZE` command to estimate the cost of each plan.
5. **Execution**: PostgreSQL selects the most efficient plan and retrieves data accordingly, using sequential scans, index scans, joins, etc.

2. Key Factors Affecting Query Performance

- **Query structure**: Efficiently structured queries often perform better.
- **Indexes**: Indexes can dramatically speed up queries but can also slow down inserts, updates and deletes.
- **Statistics**: PostgreSQL maintains statistics on tables and indexes, and the optimizer uses these to make decisions.
- **Disk I/O**: Minimize disk I/O through techniques like index optimization and caching.

Impact of Poorly Optimized Queries

- Slower Query Execution: Poorly optimized queries take longer to execute, causing delays and reducing application responsiveness.
- Increased Resource Usage: Unoptimized queries consume more CPU, memory and I/O resources, leading to potential server overload.
- System Bottlenecks: Inefficient queries can cause locks, slow down other queries, and degrade the performance of the entire database.

Interpreting and Analyzing Query Performance

1. EXPLAIN and EXPLAIN ANALYZE

PostgreSQL provides the `EXPLAIN` command, which shows the execution plan chosen by the optimizer. It outlines the series of operations that will be performed to retrieve the data. `EXPLAIN ANALYZE` executes the query and shows the actual time taken for each step.

```
EXPLAIN ANALYZE SELECT * FROM users WHERE age > 30;
```

2. Key Components of Execution Plans

- **Sequential Scan**: This means the database scans the entire table row by row, which is expensive for large tables.
- **Index Scan**: PostgreSQL uses an index to find rows, much faster for large datasets if the right index exists.
- **Nested Loop, Hash Join, Merge Join**: These are different join strategies chosen based on the size of the tables and available indexes.

3. Reading the Plan Output

- **Cost estimates**: Each operation in the plan shows two numbers *(startup cost ... total cost)*. The optimizer estimates these based on available statistics.
- **Rows**: Shows the number of rows estimated and actual.
- **Time**: `EXPLAIN ANALYZE` includes actual execution times, which can reveal performance bottlenecks.

Optimizing based on the plan:

- Replace *Sequential Scans* with *Index Scans* by adding the right indexes.

- Improve join performance by using better join types based on table sizes.

Indexing Strategies for Improved Performance

1. B-tree Indexes

The default index type in PostgreSQL. B-trees are self-balancing tree structures that maintain sorted data, making them highly efficient for equality (=) and range (<,>) queries.

2. Balanced Trees (B-trees)

Balanced trees (B-trees) ensure that the tree remains balances, meaning all leaf nodes are at the same level, making the search time logarithmic. This is crucial for maintaining performance consistency as the dataset grows.

Key Characteristics:

- **Height-balanced**: Ensures consistent search times.
- **Node structure**: Each node contains pointers to its child nodes and values in sorted order.
- **Optimized for disk access**: B-tree structures reduce the number of disk I/O operations due to the logarithmic nature.

3. Hash Indexes

Hash indexes use a hash function to distribute rows across a set of “buckets”. Hash indexing is highly efficient for equality searches but doesn’t support range queries.

4. GIN and GiST Indexes

- **GIN (Generalized Inverted Index)**: Best for full-text search or array-based queries.
- **GiST (Generalized Search Tree)**: Used for more complex data types like geometries, full-text or proximity searches.

When to use each:

- Use **B-tree** for common cases with equality and range queries.
- Use **Hash indexes** for exact matches on large datasets.

- Use **GIN** for full-text search.
- Use **GiST** for spatial and complex searches.

Optimizing Query Structure for Efficient Execution

1. **Avoid SELECT *** : Fetch only the columns you need.
2. **Use WHERE conditions efficiently**: Ensure the conditions are sargable (search argument capable), meaning they can use an index. Avoid wrapping indexed columns in functions.

```
SELECT * FROM users WHERE LOWER(name) = ‘john’; -- Cannot use
```

3. **Use Joins efficiently**: Use INNER JOIN instead of LEFT JOIN unless absolutely necessary, as LEFT JOIN can slow down queries.
4. **Limit subqueries and Correlated Subqueries**: Where possible, use JOINs instead of subqueries as they can be optimized better.

PostgreSQL Tools for Performance Monitoring and Tuning

1. **pg_stat_statements**: Track query statistics to help identify slow queries.
2. **pgbench**: A benchmarking tool to simulate different workloads.
3. **pg_top**: Monitors system resource usage by PostgreSQL processes in real-time.
4. **auto_explain**: A PostgreSQL extension that logs the execution plan of slow queries automatically.
5. **VACUUM and ANALYZE**: These commands help maintain data statistics and free up dead tuples, helping the optimizer make better decisions.

Balancing Trees and Hash Indexing

Balanced Tree Indexing (B-tree):

- Provides consistent search performance with logarithmic time complexity.
- Works well for range and equality queries.
- Tree depth remains consistent as the tree grows.

Hash Tree Indexing:

- Optimized for equality searches.
- Hashing distributes rows into buckets.
- Provides constant time complexity for exact matches but doesn't work for range queries.

Interlinking Concept for Better Understanding:

- Balanced trees (B-trees) offer a versatile index type that works well for most query types.
- Hash indexing can be faster for specific use cases but has limited application outside of equality searches.
- Understanding the execution plan help you decide when to use a specific index.
- PostgreSQL tools like EXPLAIN provide insights into where indexes are needed and whether they're being used effectively.

Important for Practice

1. Write Efficient Queries

- Fetch only the necessary columns, avoid `SELECT *` .
- Use JOINs to retrieve related data rather than nested queries.

Query with Aggregations

A query with aggregations is a SQL query that performs a calculation of summary operation over a set of rows and returns a single value or a set of values. Aggregation functions such as `SUM`, `COUNT`, `AVG`, `MAX` and `MIN` are commonly used to perform these calculations.

Aggregation is often used with the `GROUP BY` clause, which groups rows than have the same values into summary rows, enabling you to calculate aggregated values for each group.

```
SELECT c.customer_id, SUM(p.amount) AS total_payment

FROM payment p
```

```
JOIN customer c ON p.customer_id = c.customer_id

WHERE c.customer_id = 1

GROUP BY c.customer_id;
```

- Index customer_id in the payment and customer tables for faster lookup

2. Understanding and Analyzing Execution Plans

Use `EXPLAIN` and `EXPLAIN ANALYZE` to view query performance.

Key Points in the Execution Plan:

- **Seq Scan (Sequential Scan)**: Indicates the database is scanning every row in the table. It's okay for small tables but can be a performance issue on larger ones.
- **Index Scan**: If there's an index on `customer_id`, the database will use it, improving performance.
- **Cost**: Pay attention to the "total cost" and "rows" output. High cost or estimated/actual rows mismatch could mean an opportunity for optimization.

Actions:

- If a `Seq Scan` is being used, check if an index on `customer_id` exists. If not, add one:

  ```
  CREATE INDEX idx_customer_id ON customer(customer_id);
  ```

3. Indexing Strategies

a. Adding Indexes

Focus on columns that appear in `WHERE`, `JOIN`, or `ORDER BY` clauses. For example:

- **On Foreign Key Columns**: Since queries often join on foreign keys (`customer_id`, `film_id`, etc.), indexing these columns improves performance.

b. Types of Indexed to Use

- **B-tree (default)**: for equality and range queries.

- **GIN Index**: for full-text search or array operations

```
CREATE INDEX idx_title_gin ON film USING GIN (title);
```

4. Optimizing Query Structures

a. Use JOINs Instead of Subqueries

```
-- Less Efficient
SELECT *
FROM rental
WHERE customer_id = (SELECT customer_id FROM customer WHERE f

-- More Efficient
SELECT r.*
FROM rental r
JOIN customer c ON r.customer_id = c.customer_id
WHERE c.first_name = 'John';
```

- Joins are usually optimized better by the database optimizer than subqueries.

b. Avoid Complex Expressions in WHERE Clauses

Avoid expressions on indexed columns that prevent index usage.

```
-- Bad: Expression prevents index use
SELECT * FROM rental WHERE DATE(rental_date) = '2023-01-01';

-- Better: Use a range for date comparison
SELECT * FROM rental WHERE rental_date BETWEEN '2023-01-01' A
```

5. Performance Monitoring and Tuning Tools

- `pg_stat_statements`: This extension tracks all queries and their execution times. It can be used to identify slow queries.

```
CREATE EXTENSION pg_stat_statements;
```

Query it to see the most time-consuming queries:

```
SELECT query, total_time, calls, mean_time
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 5;
```

- `VACUUM` **and** `ANALYZE`: Regularly run `VACUUM` to clean up dead rows and free up space, and `ANALYZE` to update statistics that the optimizer uses.

```
VACUUM ANALYZE customer;
```