Postgres Slow Query Tune

Understanding Sequential Scan (Seq Scan) vs. Index Scan

Sequential Scan (Seq Scan)

A `Seq Scan` means that PostgreSQL is reading the entire table row by row to find the relevant data. While this approach is straightforward and necessary for certain operations (like retrieving all rows in a table), it becomes inefficient for large tables where only a subset of rows is needed. For example, in the case of filtering with `WHERE` clauses, a `Seq Scan` can result in significant performance bottlenecks.

Index Scan

An `Index Scan`, on the other hand, leverages the index structure of a table to locate specific rows more efficiently. Instead of reading every row, PostgreSQL uses the index to navigate directly to the relevant data, skipping unnecessary rows. This drastically reduces the number of disk I/O operations and speeds up the query execution time.

Key takeaway:
If you frequently encounter `Seq Scan` in `EXPLAIN ANALYZE` output for selective queries, it may indicate that the indexes are either not being used or not appropriately defined.

Index Types in PostgreSQL

PostgreSQL supports a variety of index types, each tailored for specific use cases and query patterns. Here's an overview of the main index types:

1. B-Tree Index

- **Use case:** Equality and range queries.
- **Supported operators:** `<`, `<=`, `=`, `>=`, `>` and prefix matching with `LIKE`.
- **Key features:**

- This is the default index type created by `CREATE INDEX`.
- It organizes data in a balanced tree structure, enabling efficient lookups and ordered scans.

Example:

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

2. Hash Index

- **Use case:** Equality comparisons (`=`).
- **Key features:**
- Uses 32-bit hash codes to locate data.
- Suitable for exact matches but not range queries.
- Requires enabling `enable_hashagg` and careful consideration as hash indexes are less commonly used.

Example:

```
CREATE INDEX name_hash_idx ON table_name USING HASH (column_name);
```

3. GiST (Generalized Search Tree)

- **Use case:**
- Complex data types such as geometric shapes.
- Nearest-neighbor searches and range queries.
- **Key features:**

- Provides a framework for user-defined indexing strategies, especially for multidimensional data.

Example:

```
CREATE INDEX name_gist_idx ON table_name USING GiST (column_name);
```

4. SP-GiST (Space-partitioned GiST)

- **Use case:** Geometric and spatial data.
- **Key features:**
- Optimized for partitioned search spaces (e.g., quadtrees, k-d trees).
- Efficient for nearest-neighbor and distance-based queries.

Example:

```
CREATE INDEX name_spgist_idx ON table_name USING SPGIST (column_name);
```

5. GIN (Generalized Inverted Index)

- **Use case:** Searching within arrays or documents.
- **Key features:**
- Indexes multiple components of a field (e.g., elements of an array).
- Supports full-text search but can be slower for updates due to write overhead.

Example:

```
CREATE INDEX name_gin_idx ON table_name USING GIN (column_name);
```

6. BRIN (Block Range INdex)

- **Use case:** Large, sequentially ordered datasets (e.g., time-series).
- **Key features:**
- Stores metadata for ranges of table blocks.
- Optimized for scanning ranges rather than individual rows.

Example:

```
CREATE INDEX name_brin_idx ON table_name USING BRIN (column_name);
```

Analyzing Index Usage with `pg_stat_user_indexes`

PostgreSQL provides the system view `pg_stat_user_indexes` to monitor the effectiveness of your indexes. Here's how to interpret its key columns:

- **`idx_scan`:** Number of times an index has been used for scanning. A value of `0` indicates that the index is not being utilized.
- **`idx_tup_read`:** Total rows retrieved using the index.
- **`idx_fetch`:** Number of rows fetched from the table using the index.

Example query:

```
SELECT relname, indexrelname, idx_scan, idx_tup_read, idx_tup_fetch
FROM pg_stat_user_indexes
WHERE idx_scan = 0;
```

This query highlights unused indexes, which could indicate either improper index design or queries that do not benefit from indexing.

Statistics and Autovacuum Impact

PostgreSQL relies on statistics to determine query execution plans, including whether to use an index. These statistics are periodically updated by the `autovacuum` process, but their frequency is governed by two key parameters:

`autovacuum_analyze_scale_factor`:

- Percentage of table modifications that trigger an analysis. Default: `0.1` (10% of table changes).

`autovacuum_analyze_threshold`:

- Minimum number of changes required to trigger an analysis. Default: `50`.

Manual control for statistics updates:
To check when a table was last analyzed:

```
SELECT relname, last_analyze, last_autoanalyze
FROM pg_stat_user_tables
WHERE last_autoanalyze IS NOT NULL;
```

VACUUM and ANALYZE

Running `VACUUM` alone does not update statistics. To ensure that the statistics are refreshed, you need to run `VACUUM ANALYZE`.

- **`VACUUM`:** Reclaims storage by cleaning up dead tuples.
- **`ANALYZE`:** Updates table statistics used by the query planner.
- **`VACUUM ANALYZE`:** Combines both tasks.

Command:

```
VACUUM ANALYZE;
```

Best Practices for Index Management

1. Regularly monitor index usage with `pg_stat_user_indexes`.
2. Use the appropriate index type based on your query patterns.
3. Ensure that `autovacuum` settings are optimized for your workload.
4. Periodically run `VACUUM ANALYZE` for accurate statistics.
5. Drop unused indexes to save storage and reduce maintenance overhead.

Conclusion

Indexes are a powerful feature of PostgreSQL, but they require proper understanding and management to deliver optimal performance. Using tools like `EXPLAIN ANALYZE` and monitoring views like `pg_stat_user_indexes` can help identify inefficiencies and guide you in fine-tuning your database. By leveraging the right index types and keeping statistics up-to-date, you can significantly improve query performance and reduce resource usage.