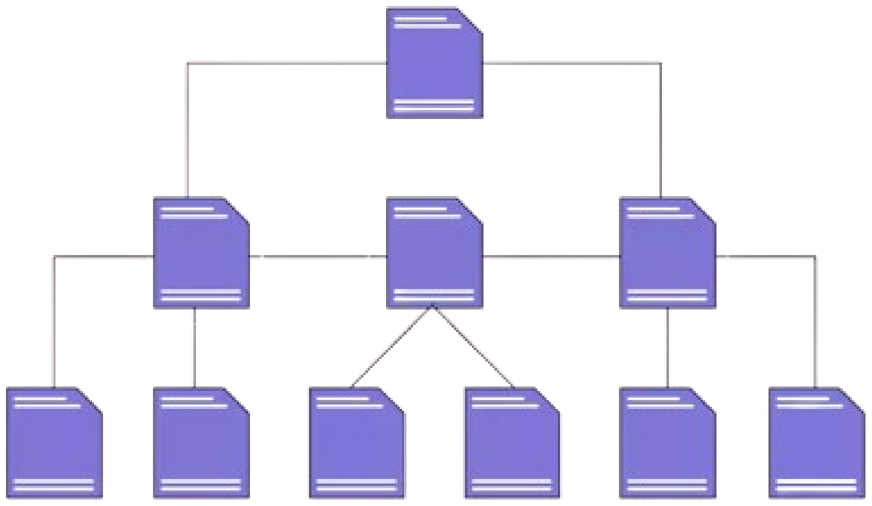
SQL
SERIES

How to Create INDEX



Indexes are essential tools in databases, ensuring efficient data retrieval.

Indexing in SQL plays a pivotal role in optimizing query performance, especially as your dataset grows.

Without the appropriate indexes, you might experience high CPU usage on your database server, slow response times, and ultimately, dissatisfied users.

Indexes are data structures that store a subset of the data in a table, organized in a way that allows the database management system to quickly locate and retrieve rows that match specific criteria.

Using the CREATE INDEX command

- To add an index to a table in MySQL or create a table index, you can use the CREATE INDEX command. The basic syntax is:

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

- To add an index to a table in MySQL or create a table index, you can use the CREATE INDEX command. The basic syntax is:

```
CREATE INDEX email_idx ON users (email);
```

Multiple-column index

- Sometimes, you might need to add an index that spans multiple columns, especially if those columns are frequently used together in queries. A multiple-column index often performs better than several single-column indexes. The syntax is:

```
CREATE INDEX index_name ON table_name (column1, column2);
```

- If your column1 and column2 contain user IDs and Organization ID, this is how your query would look:

```
CREATE INDEX user_id_and_org_id_idx ON users (user_id, org_id);
```

Unique index

- A unique index ensures that the indexed columns do not have duplicate values. This can be particularly useful for columns like email addresses, where uniqueness is crucial. The standard syntax for creating unique index is:

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

- To create unique index for the email column, with the index name users_email_uq, here's the query:

```
CREATE UNIQUE INDEX users_email_uq ON users (email);
```

Partial index or filtered index

In some cases, you might want to index only a specific portion of a string column instead of an entire table. This type of indexing is particularly useful for indexing historical data, rare or extreme values, or indexing based on status. For instance, you can create an index on the first 20 characters of a the name column that holds company names:

```
CREATE INDEX company_part_name_idx ON companies (name(20));
```

Storage order in index

Starting with MySQL version 8.x, you can specify the storage order of a column in an index. This can be beneficial if you also need to display the column in a particular order. By default, the order is ascending. In the following example, we are changing the order to descending using DESC:

```
CREATE INDEX reverse_name_idx ON companies (name DESC);
```

Functional key parts

- MySQL versions 8.0.13 and above support functional key parts. Functional key parts allow you to create an index on a function of one or more columns rather than on the columns themselves. This feature can be particularly useful in scenarios where you want to index computed values, apply functions to columns, or use expressions in your queries:

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

- In the following example, the idx_full_name index is created on the result of the CONCAT function applied to the first_name and last_name columns.

```
CREATE INDEX idx_full_name ON
employees((CONCAT(first_name, ' ', last_name)));
```

Identifying and resolve indexing issues in MySQL

Troubleshooting MySQL indexing issues can help improve the responsiveness and performance of your database queries.

Remember to carefully plan and test any index changes in a development or staging environment before applying them to a production database.

Indexing decisions should be based on your specific query patterns and use cases, and regular monitoring and maintenance are essential for maintaining optimal performance.

Missing index

Missing indexes, refer to indexes that have not been created on columns frequently used in query conditions (e.g., in the WHERE clause) or in join conditions (e.g., inJOIN operations).

When these indexes are absent, queries can become inefficient, leading to slower data retrieval and decreased overall database performance.

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

To identify missing indexes, you can use EXPLAIN or EXPLAIN ANALYZE. For the above example, let us create an index on the customer_id column.

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

Redundant index

Redundant indexes in MySQL refer to indexes that are created on the same set of columns as other existing indexes, providing little to no additional benefit in terms of query performance.

These redundant indexes can lead to increased storage requirements, slower data modification operations, and additional maintenance overhead without improving the efficiency of database queries.

Let us look at the following example where there are two indexes for the same column column1:

```
CREATE INDEX idx_column1 ON table1(column1); CREATE INDEX
idx_column1_column2 ON table1(column1, column2);
```

To resolve the above indexing issue, let us combine redundant indexes into a single, more efficient composite index (single index for table1):

```
DROP INDEX idx_column1 ON table1;
```

Composite index order

Sometimes, the order of columns in a composite (multi-column) index does not match the query conditions or the order of columns used in aJOIN operation. This can lead to suboptimal query performance.

```
SELECT * FROM products WHERE category_id = 1 AND brand_id = 2;
```

Let us create a composite index with columns in the correct order to match the existing index order. Note that this may not always be feasible, especially if you have many queries using different column orders.

```
CREATE INDEX idx_category_brand ON products(category_id, brand_id);
```

Low cardinality index

Low cardinality index is an index on a column that has a relatively small number of unique values compared to the total number of rows in a table. In other words, a low cardinality column has few distinct values, and many rows share the same value. Indexing a column with low cardinality may not improve query performance.

```
CREATE INDEX idx_status ON orders(status);
```

Consider whether indexing such a column is necessary or beneficial.

In some cases, indexing a low cardinality column may not provide significant benefits, and it may be more efficient to focus on indexing columns with higher selectivity (i.e., columns with many distinct values) or optimizing query design in other ways.

Careful consideration should be given to the specific use cases and query patterns in your database to determine whether indexing a low cardinality column is appropriate.

Index fragmentation

Index fragmentation refers to a condition where the physical storage of index data becomes disorganized or inefficient over time due to data modifications such asINSERT,UPDATE, and DELETE operations. As data in a table changes, the corresponding indexes may become less efficient, leading to performance degradation in query execution.

```
OPTIMIZE TABLE your_table;
```

Regularly optimize tables to rebuild indexes and regain performance.

Large index

Large indexes consume significant storage space and can slow down data modification operations. Evaluate whether such an index is necessary and consider the trade-offs.

```
CREATE INDEX idx_large_column ON table1(large_column);
```

Over-indexing

Having too many indexes can increase maintenance overhead and slow down data modifications.

```
CREATE INDEX idx_column1 ON table1(column1);
CREATE INDEX idx_column2 ON table1(column2);
CREATE INDEX idx_column3 ON table1(column3);
```

Review the necessity of each index and remove redundant or unused ones.

Covering index or index-only query

Covering index is a type of database index that includes all the columns required to fulfill a specific query without the need to access the actual data rows in the table.

In other words, a covering index "covers" a query by including all the information needed in the index itself, allowing the database engine to satisfy the query directly from the index structure.

This can significantly improve query performance.

```
SELECT name, email FROM customers
WHERE registration_date >= '2023-01-01';
```

Create a covering index that includes all the columns required by the query.

```
CREATE INDEX idx_registration_date ON
customers(registration_date, name, email);
```

SQL indexes for database optimization

Creating and managing indexes in MySQL is a crucial aspect of database optimization.

By understanding and utilizing the various index types and options available, you can ensure efficient data retrieval and optimal database performance.

Always remember to monitor and adjust your indexes as your data and query patterns evolve.

If you find this helpful, Repost

+ Follow for more content.

linkedin.com/in/ileonjose