Database Indexing Strategies - Part 2

BYTEBYTEGO
AUG 3, 2023 · PAID

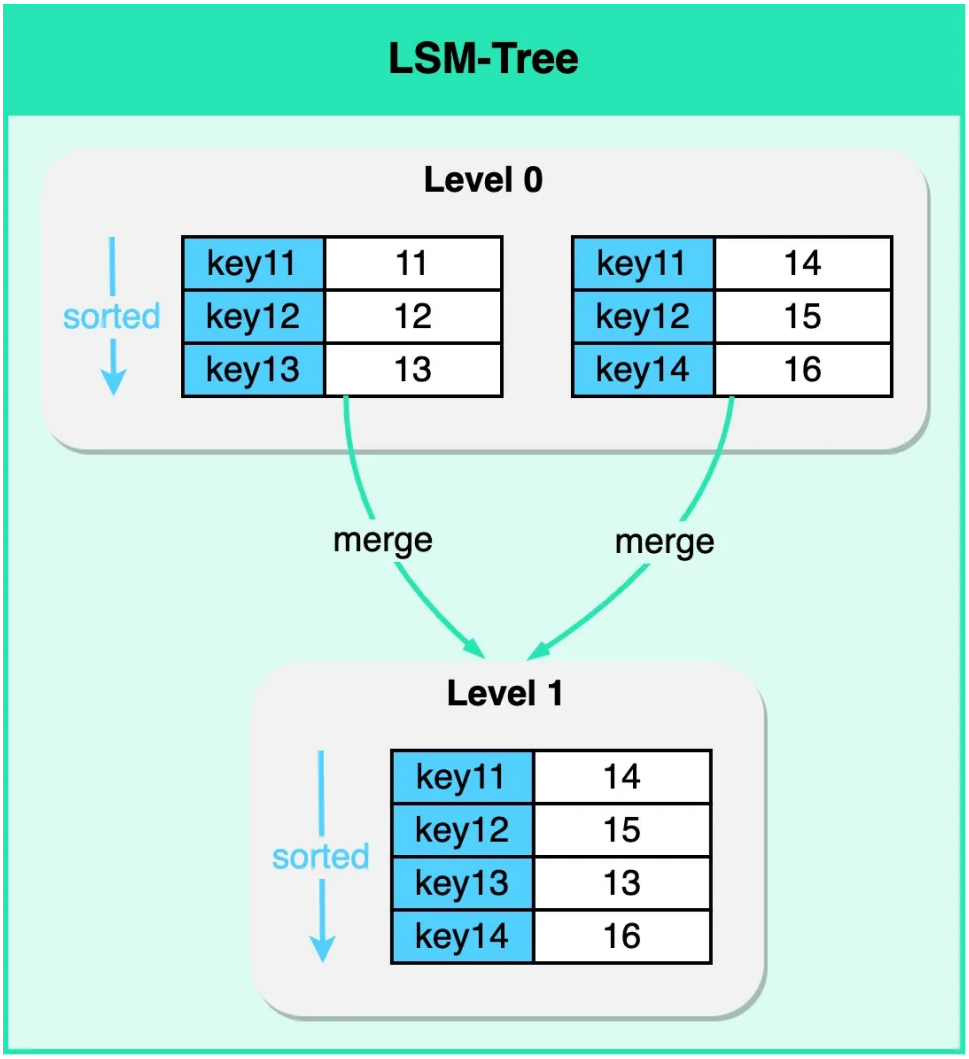
In this article, we continue the exploration of effective database indexing strategies that we kicked off in the July 6 issue. We'll discuss how indexing is used in non-relational databases and round off our discussion on general database indexing strategies with practical use cases and best practices.

Indexing in Non-relational Databases

In the July 6 issue, we focused on indexing use cases for relational databases where records are stored as individual rows. There are other popular types of databases where some forms of indexing are also used. We will briefly discuss how indexing is used in the other common form of databases - NoSQL.

NoSQL, LSM Tree, and Indexing

NoSQL databases are a broad class of database systems, designed for flexibility, scalability, and the ability to handle large volumes of structured and unstructured data. A popular data structure used in some types of NoSQL databases, notably key-value and wide-column stores, is the Log-Structured Merge-tree (LSM Tree). Unlike traditional B-Tree-based index structures, LSM Trees are optimized for write-intensive workloads, making them ideal for applications where the rate of data ingestion is high.



An LSM Tree is, in itself, a type of index. It maintains data in separate structures, each of which is a sorted tree-based index. The smaller structure resides in memory (known as a MemTable), while the larger one is stored on disk (called SSTables). Write operations are first made in the MemTable. When the MemTable reaches a certain size, its content is flushed to disk as an SSTable. The real magic of LSM Trees comes into play during read operations. While the read path is more complex due to data being spread across different structures, the LSM Tree employs techniques such as Bloom Filters and Partition Indexes to locate the required data rapidly.

Secondary Index for LSM Tree-based Databases

The LSM tree is an efficient way to perform point lookups and range queries on primary keys. However, performing a query on a non-primary key requires a full table scan which is inefficient.

This is where a secondary index is useful. A secondary index, as the name suggests, is an index that is created on a field other than the primary key field. Unlike the primary index where data is indexed based on the key, in a secondary index, data is indexed based on

non-key attributes. It provides an alternative path for the database system to access the data, allowing for more efficient processing of queries that do not involve the key.

Creating a secondary index in an LSM Tree-based database involves creating a new LSM Tree, where the keys are the values of the field on which the index is created, and the values are the primary keys of the corresponding records. When a query is executed that involves the indexed field, the database uses the secondary index to rapidly locate the primary keys of the relevant records, and then retrieves the full records from the primary index.

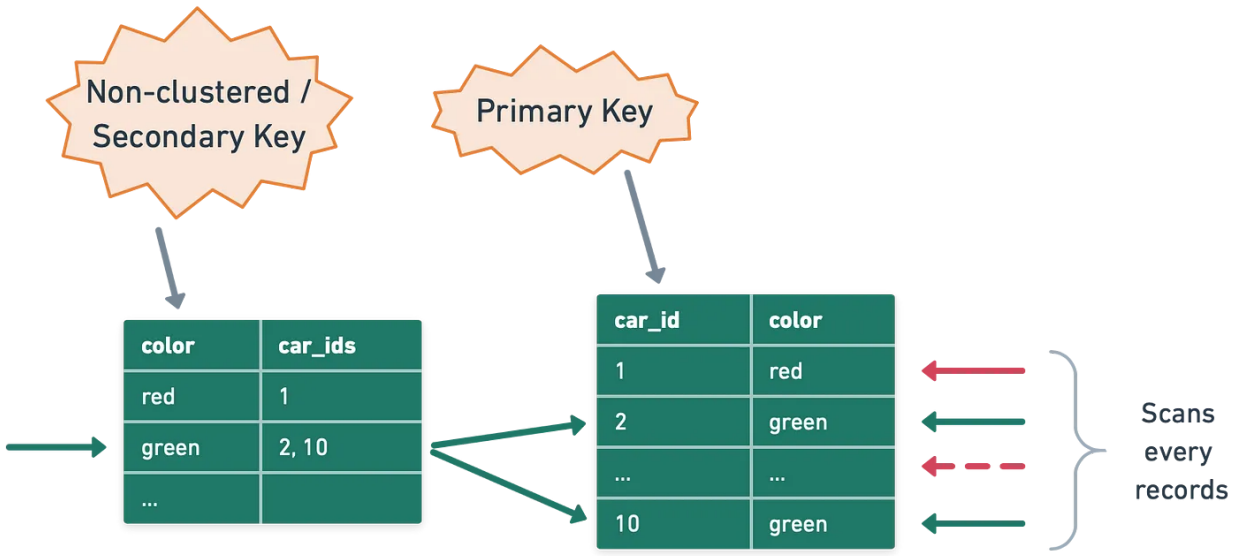
However, one of the complexities of secondary indexes in LSM Tree-based databases is handling updates. Due to the write-optimized nature of LSM Trees, data in these databases is typically immutable, which means updates are handled as a combination of a write (for the new version of a record) and a delete (for the old version). To maintain consistency, both the primary data store and the secondary index need to be updated simultaneously. This can lead to performance trade-offs and increase the complexity of maintaining index consistency.

Index Use Cases

Now that we discussed how indexes are used in different types of database systems, let’s now turn our attention to some of the common indexing use cases.

Point Lookup

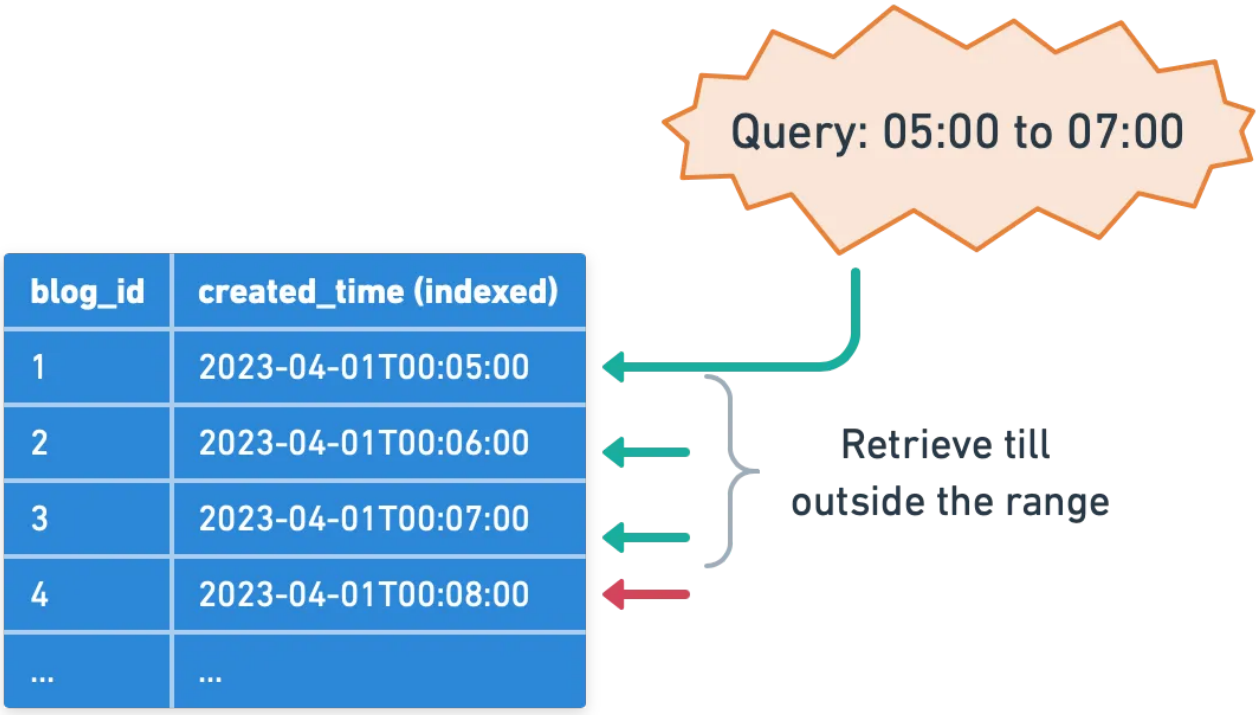
The simplest use case for an index is to speed up searches on a specific attribute or key. Let's consider an example: a car dealership has a table with columns 'car_id' and 'color'. 'Car_id' is the primary key and thus has an inherent clustered index. If we need to find a car by its 'car_id', the database can quickly locate the information.



However, what if we need to find all cars of a certain color? Without an index on the 'color' column, the database would have to scan every row in the table. This is a time-consuming process for a large table. Creating a non-clustered index on the 'color' column allows the database to efficiently retrieve all cars of a particular color, transforming what was a full table scan into a much faster index scan.

Range Lookup

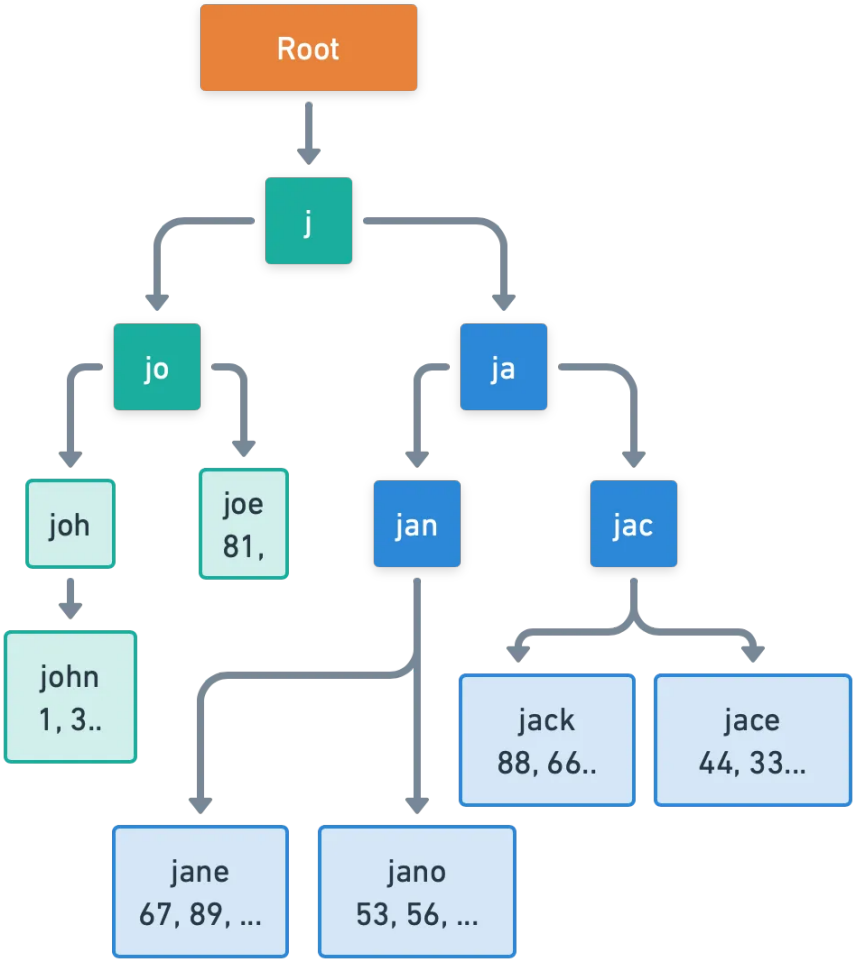
Indexes can also be used to efficiently retrieve a range of values. Consider a blog platform where 'post_id' is the primary key and 'created_time' is another attribute. Without an index, to find the 20 most recent posts, the database would need to scan all records and sort them by 'created_time'.



However, if 'created_time' is indexed, the database can use this index to quickly identify the most recent posts. This is because the index on 'created_time' stores post_id values in the order they were created, allowing the database to efficiently find the most recent entries without having to scan the entire table.

Prefix Search

Indexes are also useful for prefix searches, thanks to their sorted nature. Imagine a scenario where a search engine keeps a table of previously searched terms and their corresponding popularity scores.

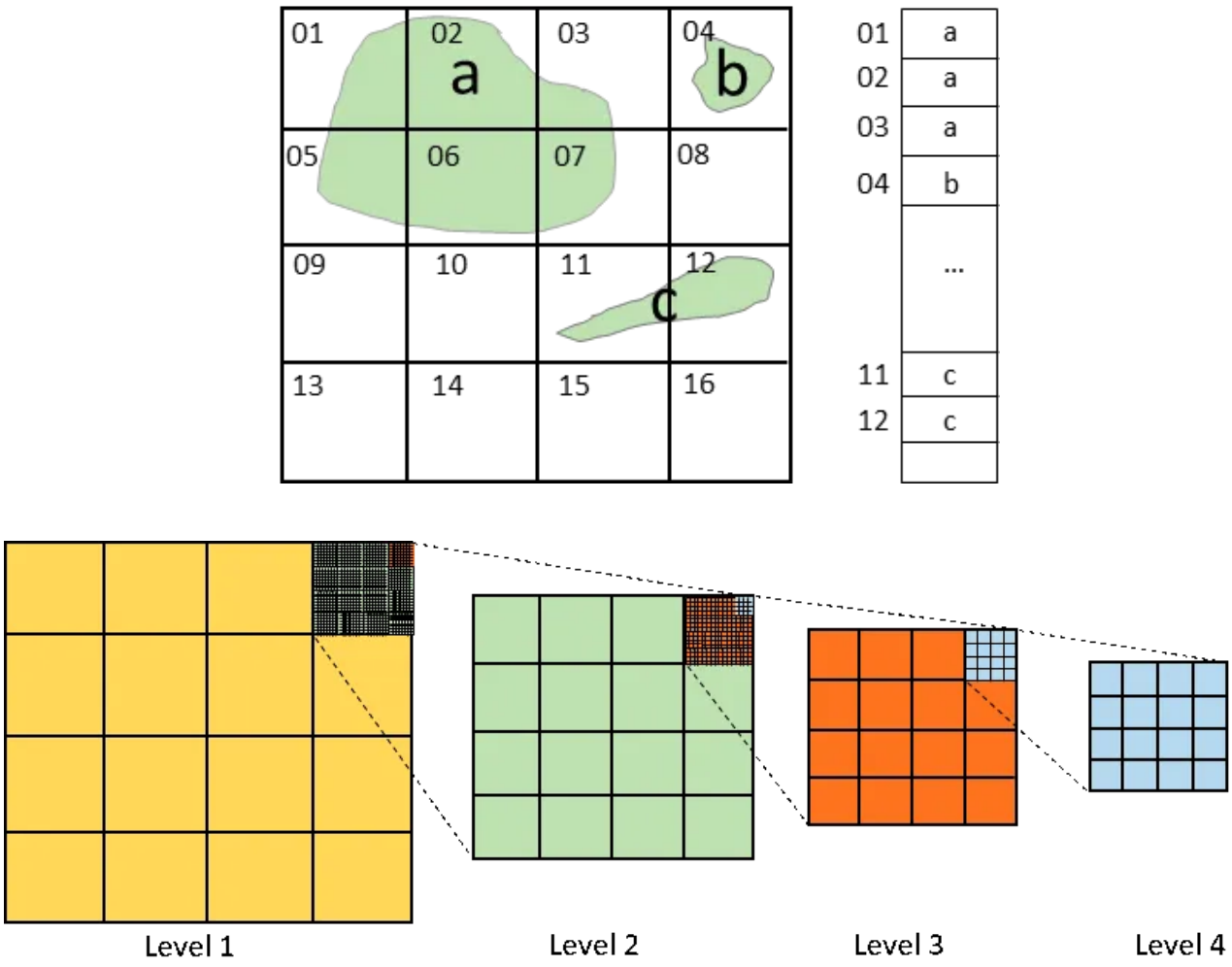


When a user starts typing a search term, the engine wants to suggest the most popular terms that start with the given prefix. A B-tree index on the search terms allows the engine to efficiently find all terms with the given prefix. Once these terms are found, they can be sorted by popularity score to provide the most relevant suggestions.

In this scenario, a prefix search can be further optimized by using a trie or a prefix tree, a special kind of tree where each node represents a prefix of some string.

Geo-Location Lookup

Geohashes are a form of spatial index that divides the Earth into a grid. Each cell in the grid is assigned a unique hash, and points within the same cell share the same hash prefix. This makes geohashes perfect for querying locations within a certain proximity.



Source: Spatial Indexing

To find all the points within a certain radius of a location, we only need to search for points that share a geohash prefix with the target location. This is a lot faster than calculating the distance to every point in the database.

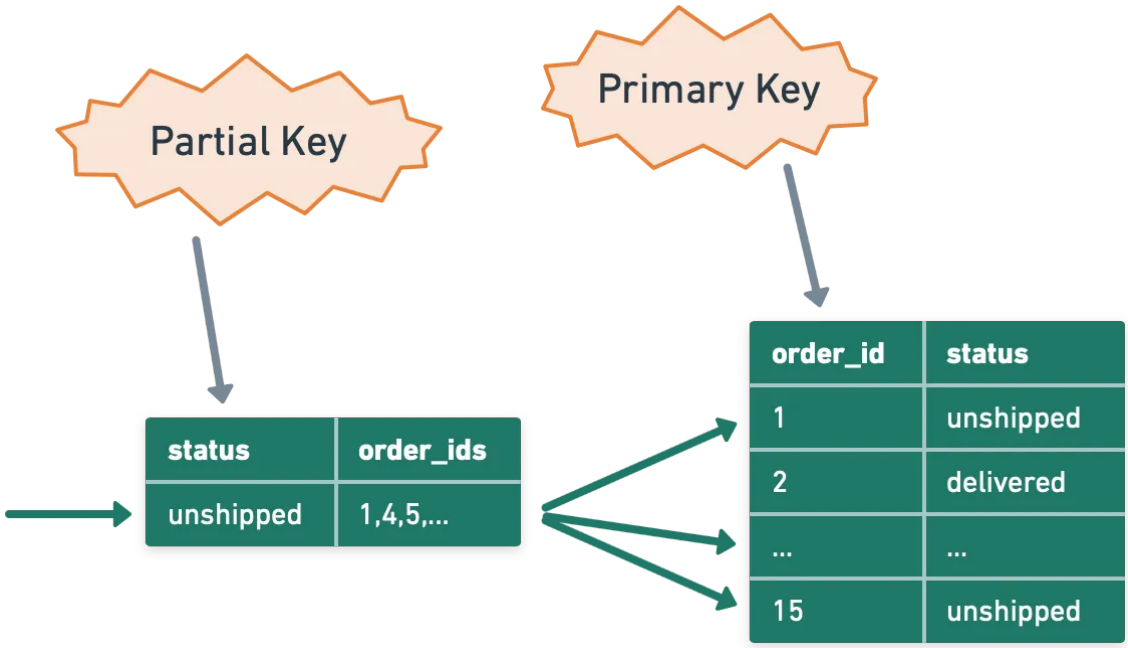
Text Search

If the application involves searching large blocks of text, a full-text index can be very beneficial. Full-text indexes enable quick text searches within a large string of text. For example, in a blogging platform, we may want to allow users to search the content of posts. A full-text index on the 'content' column can speed up these searches significantly.

Partial Index

In some cases, only a subset of a table's data is regularly queried. A partial index, also known as a filtered index, only indexes the rows that satisfy a specific condition. This can save space and improve performance by ignoring rows that aren't likely to be queried. For example, an e-commerce application might have an Orders table with a 'status' column. If

the application frequently queries for unshipped orders, a partial index on the 'status' column where 'status' equals 'unshipped' could be beneficial.'



Join Optimization

In situations where a query involves a JOIN operation between two or more tables, indexes can be useful. If there's an index on the columns being joined, the database can quickly identify matching records without scanning every row in both tables. This can greatly speed up the query execution time. For example, when joining a Customers table and an Orders table on the 'customer_id' field, having indexes on 'customer_id' in both tables will allow the database to quickly match orders with their respective customers.

```
4   SELECT Customers.customer_id, Customers.name, Orders.order_id
5   FROM Customers
6   INNER JOIN Orders ON Customers.customer_id = Orders.customer_id;
```

Order By Optimization

When the queries frequently include an ORDER BY clause on a particular column, consider indexing that column. An index will store the rows in that specific order, reducing the need for the database to sort the data whenever the query is run. For example, if we often run queries that order blog posts by their 'publish_date', an index on the 'publish_date' column will ensure that these queries run efficiently.

Group By Optimization

Similar to the ORDER BY clause, the GROUP BY clause can also benefit from indexing. If we frequently group results by a certain column, the database can utilize an index on that column to group the rows more quickly. This is especially useful when dealing with large datasets where the cost of grouping can be high.

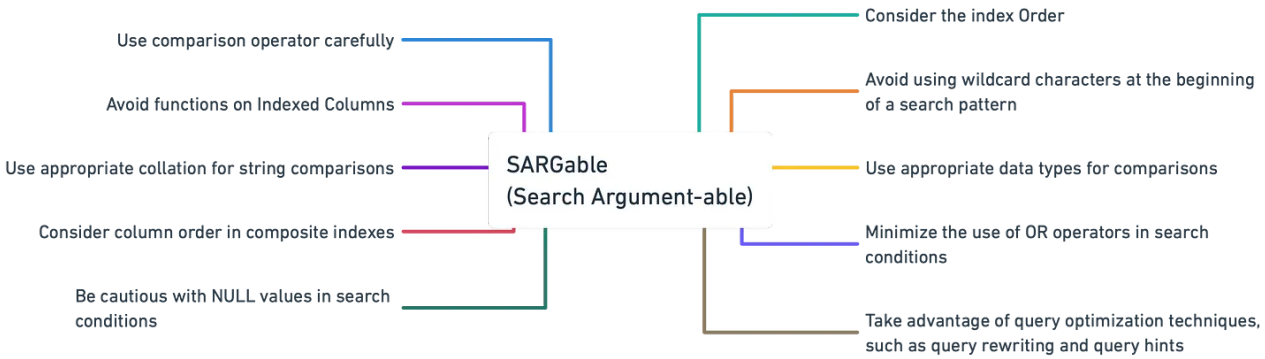
SARGable Queries

SARGable stands for "Search Argument-able," referring to a query that can take advantage of an index to speed up data retrieval. Writing SARGable queries is a crucial aspect of database optimization.

Here's why: when a query is SARGable, the database engine can use an index to find the data it needs efficiently. However, if a query is not SARGable, the database engine might need to scan every row in a table, a much slower operation, particularly with large data sets.

How do we write SARGable queries? The main idea is to write expressions that allow the database engine to search for exact values in an index. Here are a few tips to keep in mind:

- Avoid Functions on Indexed Columns: If we use a function or perform a calculation on an indexed column within the WHERE clause, the database engine can't use the index directly. For example, using `WHERE YEAR(date_column) = 2023` is not SARGable. Instead, we could rewrite this condition as `WHERE date_column BETWEEN '2023-01-01' AND '2023-12-31'`.
- Use Comparison Operators Carefully: Operators like =, >, <, >=, <=, and BETWEEN are SARGable, but NOT, LIKE (when not used with a trailing wildcard), and <> might not be, depending on the database.
- Consider the Index Order: When using a composite index, remember that the order of the columns in the WHERE clause matters. If the index is on columns (A, B, C), a query filtering on column A or columns A and B can utilize the index, but a query filtering on B and C cannot.



Indexing Best Practices

With all the necessary background information, let's apply what we have learned and distill them into a set of best practices.

Navigating the world of database indexing can be a daunting task, given the wide array of considerations and trade-offs involved in choosing the right index. Below are some of the best practices we came up with.

Determine Our Needs

Before creating any indexes, we need to have a clear understanding of the application's requirements. This involves a thorough analysis of the application's workload. Identify the most common queries and understand how frequently they are used. In addition, determine the application's read-to-write ratio. Indexing improves read performance but can slow down write performance. If the application performs more writes than reads, too many indexes could adversely affect overall performance. Similarly, if the application is read-heavy, appropriate indexing can significantly enhance efficiency.

Choose the Right Columns for Indexing

The decision of which columns to index should be based on their usage in queries. Indexes should ideally be created on columns that are frequently used in WHERE clauses, ORDER BY clauses, JOIN conditions, or used for sorting and grouping data.

Columns with a high degree of uniqueness are ideal candidates for indexing. Indexes on such columns allow the database engine to quickly filter out a majority of the data. It leads to faster query results. Avoid indexing columns with many null values or those that have a lot of similar values.

Weigh the Cost of Updates to Indexed Columns

While indexes speed up data retrieval, they slow down data modification. This is because each time data is added, deleted, or modified, the corresponding indexes need to be updated as well. If a column is frequently updated, the overhead of updating the index might negate the performance benefits gained during data retrieval. If the index update time outweighs the time saved during data retrieval, it might not be worth it to maintain the index.

Limit the Number of Indexes

While indexes are beneficial for query performance, having too many can negatively impact the performance of write operations and consume more disk space. Each time data is inserted or updated, every index on the table must be updated. The cost of maintaining the index might outweigh the performance benefits it provides. It is important to maintain a healthy balance and limit the number of indexes based on the nature of the workload. Utilize monitoring tools provided by the database system to identify if the update time on the indexes is increasing disproportionately.

Use Composite Indexes Effectively

Composite indexes, which are made up of two or more columns, can be very beneficial for complex queries that involve multiple columns in the WHERE clause. The order of the columns in the composite index is critical. As a general rule of thumb, it should be based on the cardinality of the columns, with the column having the highest number of distinct values appearing first in the index. This order allows the database engine to efficiently filter out unneeded data. For example, if we are creating a composite index on "CustomerName" and "Country" columns in a 'Orders' table, and there are fewer distinct countries than customer names, the index should be (Country, CustomerName).

It is important to note that this guideline is a general rule of thumb, and it is not always correct. Verify with the optimizer that it indeed uses the composite index as intended.

Leverage Covering Indexes

A covering index includes all the columns that a query needs, both in the WHERE clause and the SELECT list. This means that the database engine can locate all the required data within the index itself, without having to perform additional lookups in the main table. This results in a significant performance boost because accessing an index is typically faster than accessing the table data. Consider using covering indexes for frequently used, read-intensive queries.

Regularly Monitor and Optimize the Indexes

Indexes are not a set-it-and-forget-it part of the database. As the data grows and changes, the indexes need to be monitored and optimized. Over time, as data is added, updated, and deleted, indexes can become fragmented, which can negatively impact their performance. Regularly performing index maintenance tasks, such as rebuilding or reorganizing fragmented indexes, can help ensure that they continue to provide optimal performance. Database tools such as SQL Server's Database Engine Tuning Advisor or MySQL's OPTIMIZE TABLE command are some examples of tools to use. Monitoring logs like MySQL's slow query log is also important in detecting issues early.

Drop Unused Indexes

Not all indexes end up being used as intended. Some may be rarely used, or not at all. Such indexes impose unnecessary overhead on write operations and waste storage space. Use the database's built-in tools to monitor index usage, and do not hesitate to drop indexes that are no longer serving their purpose. In PostgreSQL, for instance, we can use the pg_stat_user_indexes view to track index usage.

Conclusion

Database indexing is a critical part in optimizing database efficiency. It's a key component in the balancing act between speed of data retrieval and the performance of write operations.

However, indexing is not a one-size-fits-all solution. Careful design, regular monitoring and maintenance are vital components in maximizing the benefits of the indexing strategies. Recognizing the indexing techniques specific to different databases can significantly improve the data operations.

A well-implemented indexing strategy is fundamental to a high-performing database. Mastering the art of database indexing is an indispensable skill for anyone building large-scale data-driven applications.

127 Likes · 8 Restacks

3 Comments

Write a comment...

Ujjal Dutta Aug 9

The example under “use composite index effectively” is contradictory to the general rule of thumb prescribed. The index should be created on (CustomerName, Country) since CustomerName has higher cardianality I.e. more distinct values as compared to Country names

LIKE (4) REPLY SHARE

1 reply

Indian Aug 4

Thank You team, great article and information.

As requested earlier in most of the post, will request again to provide the reference link for readers.

Few Questions:

1. Mostly in most of use cases are, people have few sql queries which they think are really slow, Please if possible suggest us the steps we should see in Query Execution Plan and try to fix

those slow queries ?

2. In terms of No-SQL databases, those can be scaled horizontally, how indexing happens and get maintained in those scenarios where data is indexed and mapped across different boxes ?

3. Is there any benchmarking article we can refer in case of write / read speed on SQL and NoSQL data bases ?

4. It is said that indexing in No-SQL databases is not born given, but it is mainly for SQL databases ? How we should choose our indexing strategies between SQL and No SQL databases ?

Thanks to the Entire Team for this useful and helpful Information once again.

LIKE REPLY SHARE

1 more comment...