2023-10-30

应用系统体系架构 — 作业6

学号：521030990006

姓名：VAHAGN GHAZARYAN

## **A.When executing queries on a clustered index, especially range queries, its execution efficiency will be very high. Why?**

## When executing range queries on a clustered index:

## Data is stored sequentially based on the index key.

## The B-Tree structure allows for efficient disk reads with many keys in a single node.

## Once the range start is found, subsequent data is read sequentially, minimizing disk seeks.

## The ordered B-Tree ensures related data is close together, reducing disk I/O operations.

## **B.If you want to store your book cover in Base 64 format in MySQL, should you store it in VARCHAR type or LONGBLOB format in the design of the data table? Please explain your reasons in detail.**

## It's better to use LONGBLOB over VARCHAR.

## VARCHAR has a maximum length of 65,535 characters. Base64 encoding increases the size of binary data by about 33%, so even a small image could exceed the VARCHAR limit once encoded.

## LONGBLOB can store up to 4GB, giving ample space for encoded images.

## Using VARCHAR implies character data with semantic meaning, but Base64-encoded image data doesn't fit this description.

## Storing large blocks of data in VARCHAR could affect performance, especially during string operations.

## LONGBLOB is specifically designed for storing large amounts of binary data.

## **When you build a composite index on your book table to speed up queries involving multiple fields, use SQL statements to give the way you build the composite index, and explain in detail why you built such an index, including field order, ascending and descending sorting. How are factors such as determined?**

## ***CREATE INDEX idx\_composite ON books (type ASC, author ASC, price DESC);***

## I've chosen type, author, and price for the composite index. This index would efficiently support queries that filter or sort based on the genre/type of the book, the author, and the price.

## **type**: It's common for users to search for books of a specific genre or type. By placing type first in the index, queries that filter by book type will be faster.

## **author**: After narrowing down a genre, users might want to search for books by a specific author. Having author next in the index supports this query pattern.

## **price**: Users might further want to sort or filter books based on price, especially within a specific genre and author. Hence, price is added last in the index.

## **type and author** are sorted in ascending order because it's typical to retrieve such categorical and textual data in alphabetical order. **price** is sorted in descending order, assuming users might be interested in viewing more expensive books first within a category and author.

## When making a composite index, I think about how users search, so I focus on book type, then author, and maybe price. "Type" and "author" are chosen because they help narrow down results quickly and are faster to check than longer details. But, if these details change a lot, it might slow down the system, though they usually change less than things like "inventory" or "description".

## **Do you think it is better to use auto-increment primary key or UUID as the primary key of your order table ? Why ?**

## If the database is distributed or there's a potential need for merging data across systems in the future, a UUID might be more suitable due to its global uniqueness. Additionally, for security, UUIDs offer an advantage as their randomness makes them harder to guess, addressing potential security concerns. However, if performance and storage efficiency are primary considerations, then an auto-increment primary key, with its smaller size and better indexing performance, would be the preferred choice. It’s simple and straightforward. So I think it is better to keep the orders table use auto increment .

## **Please search the references and summarize the main differences between the two storage engines InnoDB and MyISAM .**

## **Transaction**

## InnoDB: Supports ACID-compliant transactions, meaning can group multiple SQL statements into a single transaction.

## MyISAM: Doesn't support transactions. Each SQL statement is treated as a single transaction.

## **Locking**

## InnoDB: Employs row-level locking, which means higher concurrency and better performance for write-intensive workloads.

## MyISAM: Uses table-level locking, which can be a bottleneck for operations that modify data.

## **Foreign key constraints**

## InnoDB: Features foreign key constraints, ensuring data integrity and relationships between tables.

## MyISAM: Lacks support for foreign key constraints.

## Storage:

## **Storage Space**

## InnoDB: Generally consumes more disk space than MyISAM due to its transaction logs.

## MyISAM: Typically uses less disk space.

## **Full-text Search**

## InnoDB: Added support for full-text search in MySQL 5.6 and later.

## MyISAM: Has native support for full-text search.

## **Compression**

## InnoDB: Supports data compression.

## MyISAM: Doesn't support data compression in the same way; though, there are workarounds.

## **Backup**

## InnoDB: Can leverage consistent online backups, especially with tools like innobackupex.

## MyISAM: Backups are typically more challenging, especially for live systems, because of the table-locking mechanism.