MYSQL

CHO NGƯỜI ĐI LÀM

Các chiến lược tối ưu MySQL cơ bản

Nguyễn Thế Huy
huynt57@gmail.com
https://huynt.dev

Lời mở đầu

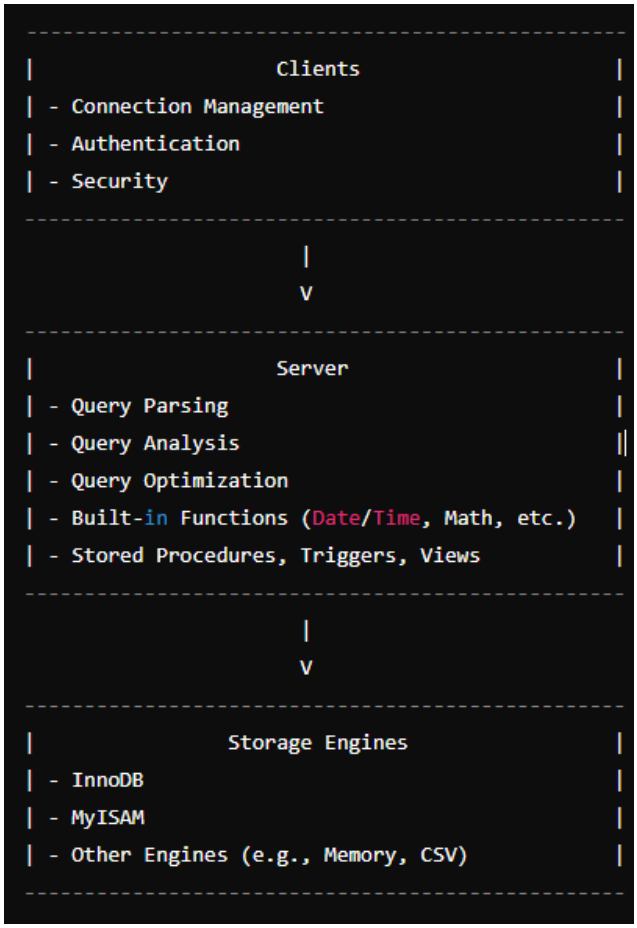
Chào bạn, và cám ơn bạn đã đọc ebook này của mình.
Mình là Huy, hiện tại mình là Technical Leader tại Công ty Cổ phần Giao hàng Tiết kiệm. Ngoài ra, mình còn là Moderator của Group Laravel Việt Nam, một cộng đồng Laravel lớn trên Facebook với gần 30.000 thành viên.
Ebook này của mình nhằm mang đến kiến thức về các chiến lược tối ưu MySQL cơ bản cho mọi người, phân tích các best practices phổ biến khi bạn thiết kế và làm việc cùng MySQL. Dù bạn là người mới bắt đầu hay đã có kinh nghiệm, mình tin những kiến thức trong ebook này sẽ giúp bạn rất nhiều trong việc chuẩn bị nền tảng kiến thức để làm việc với hệ quản trị cơ sở dữ liệu nổi tiếng này.
Chúc bạn có những giây phút thú vị, bổ ích khi đọc ebook.

Kiến trúc MySQL

Trước khi bắt tay tìm hiểu các chiến lược tối ưu MySQL, chúng ta hãy cùng nhau làm quen với các kiến trúc cơ bản của hệ quản trị cơ sở dữ liệu này. Section này sẽ cung cấp cho bạn những góc nhìn tổng quan về kiến trúc MySQL, cũng như giúp bạn hiểu hơn về cách mà MySQL hoạt động.

Các layer trong kiến trúc MySQL

MySQL sở hữu một kiến trúc nhiều lớp. Cách thiết kế này không hẳn là hoàn hảo, nhưng nó mang lại cho MySQL khả năng linh hoạt trong nhiều tình huống khác nhau. Hình vẽ dưới đây miêu tả về kiến trúc của MySQL để bạn dễ hình dung:



Lớp trên cùng: Khách hàng (Clients)

Lớp trên cùng này chứa các dịch vụ liên quan đến giao tiếp - kết nối của MySQL: xử lý kết nối, xác thực, bảo mật…

- **Xử lý kết nối**: Quản lý các kết nối từ ứng dụng hoặc người dùng, đảm bảo rằng mỗi kết nối được quản lý một cách hiệu quả.
- **Xác thực**: Kiểm tra thông tin xác thực của người dùng hoặc ứng dụng để đảm bảo rằng chỉ những người dùng hợp lệ mới được phép truy cập cơ sở dữ liệu.
- **Bảo mật**: Bảo vệ dữ liệu và các kết nối khỏi các truy cập trái phép và các mối đe dọa bảo mật khác.

Lớp thứ hai: Tầng SQL (SQL Layer)

Phần lớn những thứ thú vị nhất của MySQL nằm ở đây, bao gồm các luồng xử lý cho việc phân tích cú pháp truy vấn, tối ưu hóa, và tất cả các built-in function (ví dụ: ngày tháng, thời gian, toán học, …):

- **Phân tích cú pháp truy vấn**: Chuyển đổi các truy vấn SQL thành các cấu trúc nội bộ (cây phân tích) để máy chủ có thể hiểu và xử lý.
- **Phân tích và tối ưu hóa**: Áp dụng nhiều kỹ thuật tối ưu hóa để xác định cách tốt nhất để thực thi truy vấn, bao gồm việc chọn index phù hợp và xác định thứ tự đọc các bảng.
- **Thực thi truy vấn**: Thực hiện truy vấn và trả về kết quả cho người dùng hoặc ứng dụng.
- **Các hàm tích hợp sẵn**: Cung cấp các hàm tích hợp cho các phép toán ngày tháng, thời gian, toán học, và mã hóa.

Lớp thứ ba: Các engine lưu trữ (Storage Engines)

Đây là nơi dữ liệu được lưu trữ và truy xuất. Các storage engine như InnoDB, MyISAM … cung cấp các cách khác nhau để quản lý dữ liệu. Mỗi engine lưu trữ có những ưu và nhược điểm riêng và được thiết kế để phù hợp với các loại công việc khác nhau. Tầng này chứa API của storage engine, giúp cho chúng trở nên trong suốt tại tầng truy vấn (có nghĩa là các storage engine khác nhau đều có thể giao tiếp được với tầng SQL). Điều đó mang đến tính linh hoạt cao cho MySQL, vì bạn có thể sử dụng nhiều storage engine khác nhau tuỳ từng nhu cầu của mình.

Quá trình tối ưu hóa và thực thi

- **Phân tích và tối ưu hóa**: MySQL phân tích cú pháp truy vấn để tạo ra một cấu trúc nội bộ (cây phân tích) và sau đó áp dụng nhiều kỹ thuật tối ưu hóa. Các kỹ thuật này bao gồm việc thay đổi cấu trúc truy vấn, xác định thứ tự đọc các bảng, chọn Index sử dụng, và các tối ưu hóa khác. MySQL có thể nhận các gợi ý tối ưu hóa từ người dùng qua các từ khóa đặc biệt trong truy vấn và cung cấp thông tin về các quyết định tối ưu hóa thông qua lệnh `EXPLAIN`.
- **Thực thi truy vấn**: Optimizer không quan tâm storage engine cụ thể nào đang được sử dụng, nhưng storage engine ảnh hưởng đến cách MySQL tối ưu hóa truy vấn. Optimizer sẽ hỏi storage engine về các khả năng tối ưu mà storage engine có thể cung cấp, cũng như các chi phí khi áp dụng các tối ưu này. Ví dụ, một số storage engine có thể hỗ trợ một số loại index đặc biệt hữu ích trong một số truy vấn cụ thể. Mình sẽ bàn chi tiết hơn về điều này trong section tối ưu hóa schema và index.

So sánh InnoDB và MyISAM trong MySQL

MySQL hỗ trợ nhiều loại storage engine, trong đó hai loại phổ biến nhất là InnoDB và MyISAM. Mỗi loại có những đặc điểm, ưu điểm và nhược điểm riêng, phù hợp với các tình huống và nhu cầu khác nhau.

1. Khả năng hỗ trợ giao dịch (Transactions)

- **InnoDB**: Hỗ trợ transaction với các đặc tính ACID (Atomicity, Consistency, Isolation, Durability). Điều này đảm bảo tính nhất quán dữ liệu và bảo vệ dữ liệu trong trường hợp có lỗi hoặc sự cố hệ thống.
- **MyISAM**: Không hỗ trợ transaction . Dữ liệu có thể bị hỏng nếu xảy ra lỗi trong quá trình cập nhật.

2. Khóa (Locking)

- **InnoDB**: Hỗ trợ khóa cấp hàng (row-level locking), cho phép nhiều giao dịch cập nhật cùng lúc mà không gây xung đột, nâng cao hiệu suất trong môi trường có nhiều truy vấn đồng thời.
- **MyISAM**: Sử dụng khóa cấp bảng (table-level locking). Khi một hàng bị khóa để cập nhật, toàn bộ bảng bị khóa, dẫn đến hiệu suất thấp hơn trong trường hợp có nhiều truy vấn đồng thời.

3. Khôi phục dữ liệu sau sự cố (Crash Recovery)

- **InnoDB**: Có khả năng khôi phục dữ liệu sau sự cố nhờ vào tính năng ghi nhật ký (redo logs) và các cơ chế khôi phục tự động.
- **MyISAM**: Khả năng khôi phục dữ liệu kém hơn. Sau một sự cố, bảng MyISAM có thể cần phải kiểm tra và sửa chữa thủ công.

4. Hỗ trợ khóa ngoại (Foreign Key Constraints)

- **InnoDB**: Hỗ trợ khóa ngoại, giúp duy trì tính toàn vẹn của dữ liệu giữa các bảng liên quan.
- **MyISAM**: Không hỗ trợ khóa ngoại.

5. Hiệu Suất Truy Vấn Đọc

- **InnoDB**: Hiệu suất đọc tốt, nhưng không nhanh bằng MyISAM trong một số trường hợp cụ thể vì cần phải quản lý thêm các tính năng như transaction và khóa cấp hàng (row-level locking).
- **MyISAM**: Hiệu suất đọc cao hơn cho các truy vấn chỉ đọc do không phải quản lý các tính năng liên quan đến transaction.

6. Kích thước tập tin chỉ mục (Index File Size)

- **InnoDB**: Index có thể lớn hơn vì lưu trữ thông tin về giao dịch và khóa.
- **MyISAM**: Index nhỏ gọn hơn, giúp tăng tốc độ truy vấn tìm kiếm.

7. Tìm kiếm toàn văn bản (Full-Text Search)

- **InnoDB**: Bắt đầu từ MySQL 5.6, InnoDB đã hỗ trợ tìm kiếm toàn văn bản, nhưng vẫn không mạnh mẽ như MyISAM.
- **MyISAM**: Hỗ trợ tìm kiếm toàn văn bản từ lâu và hiệu suất tìm kiếm toàn văn bản thường tốt hơn InnoDB.

8. Kích thước bảng tối đa

- **InnoDB**: Hỗ trợ kích thước bảng lên tới 64TB.
- **MyISAM**: Hỗ trợ kích thước bảng lên tới 256TB, phụ thuộc vào kích thước tập tin và hệ thống tệp.

9. Backup và khôi phục dữ liệu

- **InnoDB**: Hỗ trợ sao lưu và khôi phục dữ liệu nóng (hot backup) nhờ tính năng transaction và nhật ký redo.
- **MyISAM**: Chỉ hỗ trợ sao lưu và khôi phục dữ liệu khi hệ thống không có hoạt động ghi (cold backup).

Bonus: Có nên sử dụng ràng buộc (Constraint) ở tầng cơ sở dữ liệu ?

Ràng buộc (constraints) trong cơ sở dữ liệu là các quy tắc được áp dụng trên các bảng để đảm bảo tính toàn vẹn và nhất quán của dữ liệu. Các ràng buộc này được quản lý trực tiếp trên MySQL. Một số ràng buộc phổ biến như PRIMARY KEY, FOREIGN KEY, UNIQUE …. Trong phần này, mình bàn về việc có nên sử dụng ràng buộc liên quan đến FOREIGN KEY hay không.
Đối với cá nhân mình, bạn không nhất thiết phải cài đặt ràng buộc FOREIGN KEY. Một số hạn chế của nó bên cạnh lợi ích như sau:

Giảm hiệu suất cập nhật:

Ràng buộc FOREIGN KEY có thể làm giảm hiệu suất ghi chép dữ liệu, đặc biệt khi có nhiều thao tác ghi hoặc cập nhật dữ liệu vì cơ sở dữ liệu phải thực hiện các kiểm tra bổ sung để duy trì tính toàn vẹn tham chiếu.

Phức tạp trong quản lý:

Việc thêm, sửa đổi hoặc loại bỏ ràng buộc FOREIGN KEY có thể phức tạp, đặc biệt khi cơ sở dữ liệu lớn hoặc có nhiều mối quan hệ tham chiếu phức tạp.

Giới hạn tính linh hoạt:

Ràng buộc FOREIGN KEY có thể làm giảm tính linh hoạt của cơ sở dữ liệu, đặc biệt khi có các yêu cầu thay đổi cấu trúc dữ liệu hoặc nghiệp vụ thường xuyên. Xu hướng hiện tại cũng chấp nhận việc dư thừa dữ liệu nhiều hơn, nên việc áp dụng ràng buộc khoá ngoại chặt chẽ cũng không thực sự cần thiết.

Khó khăn trong phát triển và kiểm tra:

Trong quá trình phát triển, ràng buộc FOREIGN KEY có thể làm cho việc nhập dữ liệu thử nghiệm trở nên khó khăn hơn, vì bạn phải đảm bảo rằng tất cả các mối quan hệ tham chiếu đều hợp lệ.

Với các vấn đề trên, cá nhân mình khuyên bạn không cần triển khai ràng buộc FOREIGN KEY trong thực tế (trừ khi bạn thực sự cần sử dụng tới nó).

Bonus: Có nên sử dụng Stored Procedure trong MySQL?

Stored procedure (thủ tục lưu trữ) là các đoạn mã SQL được lưu trữ và thực thi trên máy chủ cơ sở dữ liệu. Chúng có thể chứa các lệnh SQL để thực hiện các tác vụ cụ thể, và có thể được gọi từ các ứng dụng hoặc từ chính cơ sở dữ liệu. Dưới đây là một số lợi ích và hạn chế của việc sử dụng stored procedure trong MySQL để giúp bạn quyết định liệu có nên sử dụng chúng hay không:

Lợi ich

Hiệu suất:

Stored procedure có thể cải thiện hiệu suất bằng cách giảm tải công việc xử lý từ ứng dụng sang máy chủ cơ sở dữ liệu. Các thủ tục này được biên dịch một lần và có thể được thực thi nhiều lần mà không cần phải biên dịch lại.

Tính nhất quán và tái sử dụng:

Bạn có thể viết các đoạn mã phức tạp một lần dưới dạng stored procedure và sử dụng lại chúng ở nhiều nơi khác nhau, giúp đảm bảo tính nhất quán trong việc thực hiện các tác vụ.

Bảo mật:

Stored procedure có thể giúp tăng cường bảo mật bằng cách giới hạn quyền truy cập trực tiếp vào các bảng dữ liệu. Người dùng có thể được cấp quyền thực thi stored procedure mà không cần quyền truy cập vào dữ liệu thô.

Giảm lưu lượng mạng:

Khi sử dụng stored procedure, ứng dụng chỉ cần gửi lệnh gọi thủ tục đến máy chủ cơ sở dữ liệu thay vì gửi toàn bộ câu lệnh SQL, giảm bớt lưu lượng mạng.

Hạn chế

Khả năng phát triển và bảo trì:

Việc phát triển và bảo trì stored procedure có thể phức tạp hơn so với mã ứng dụng, đặc biệt khi bạn có nhiều thủ tục lưu trữ phức tạp.

Độ phụ thuộc vào cơ sở dữ liệu:

Stored procedure làm tăng độ phụ thuộc vào hệ quản trị cơ sở dữ liệu cụ thể, khiến cho việc di chuyển sang hệ quản trị khác khó khăn hơn.

Khó khăn trong Debug:

Việc gỡ lỗi stored procedure có thể khó khăn hơn so với mã ứng dụng, vì các công cụ hỗ trợ gỡ lỗi cho SQL thường không mạnh mẽ bằng các công cụ phát triển phần mềm.

Khi cân nhắc giữa ích lợi và hạn chế của Stored Procedure, trong thực tế, mình cũng hiếm khi sử dụng nó (trừ một số trường hợp đặc biệt, hoặc migrate dữ liệu). Mình vẫn ưu tiên việc kiểm soát ở tầng ứng dụng các truy vấn SQL hơn là sử dụng Stored Procedure.

Transaction (Giao dịch)

Transaction là một thuật ngữ đã quá quen thuộc khi bạn làm việc với cơ sở dữ liệu. Một transaction là một nhóm các câu lệnh SQL được xử lý atomic (nguyên tử), như một đơn vị công việc duy nhất. Trong quá trình xử lý tập các câu lệnh SQL trong một transaction, nếu có bất kỳ câu lệnh nào không thể thực hiện được vì lý do nào đó (như hệ thống gặp sự cố), thì sẽ không có câu lệnh SQL nào được thực thi trong tập các câu lệnh trên. Điều này còn được mô tả bằng thuật ngữ “tất cả hoặc không gì cả” (Hoặc là tất cả các câu lệnh thành công, hoặc sẽ không có câu lệnh nào được thực thi).

Ví dụ

Một ứng dụng ngân hàng là ví dụ kinh điển về lý do tại sao transaction là cần thiết (Tiền thì lúc nào cũng quan trọng mà :D) . Hãy tưởng tượng một cơ sở dữ liệu ngân hàng với hai bảng: `checking` (Tài khoản giao dịch) và `savings` (Tài khoản tiết kiệm). Để chuyển 200.000 từ tài khoản giao dịch của Huy sang tài khoản tiết kiệm của anh ấy, bạn cần thực hiện ít nhất ba bước:

- Đảm bảo rằng số dư tài khoản giao dịch của anh ấy lớn hơn 200.000.
- Trừ 200.000 từ số dư tài khoản giao dịch của anh ấy.
- Cộng 200.000 vào số dư tài khoản tiết kiệm của anh ấy.

Toàn bộ hoạt động này nên được bao bọc trong một transaction để nếu bất kỳ bước nào bị lỗi, các bước đã hoàn thành có thể được hoàn tác (rollback).

Bạn bắt đầu một transaction với câu lệnh `START TRANSACTION` và sau đó xác nhận apply các thay đổi bằng lệnh `COMMIT` hoặc hủy bỏ các thay đổi với `ROLLBACK`. SQL cho transaction này của chúng ta có thể trông như sau:

```
START TRANSACTION;
SELECT balance FROM checking WHERE customer_name = “Huy”;
UPDATE checking SET balance = balance - 200000 WHERE customer_name = “Huy”;
UPDATE savings SET balance = balance + 200000 WHERE customer_name = “Huy”;
```

```
COMMIT;
```

Tính chất ACID

Một transaction sẽ không được coi là hoàn thiện nếu nó không đạt đủ tính chất ACID. ACID là viết tắt của các đặc tính:

- **Atomicity (Tính Nguyên Tử)**: Transaction phải hoạt động như một đơn vị công việc không thể chia nhỏ, toàn bộ giao dịch hoặc được áp dụng hoàn toàn hoặc sẽ không thành công toàn bộ. Ví dụ**:** Trong một hệ thống ngân hàng, khi chuyển tiền từ tài khoản A sang tài khoản B, transaction bao gồm hai bước: trừ tiền từ tài khoản A và cộng tiền vào tài khoản B. Tính nguyên tử đảm bảo rằng hoặc cả hai bước đều được thực thi hoặc không bước nào được thực thi.
- **Consistency (Tính Nhất Quán)**: Tính nhất quán đảm bảo rằng một transaction sẽ đưa cơ sở dữ liệu từ một trạng thái nhất quán này sang một trạng thái nhất quán khác. Tính nhất quán duy trì các ràng buộc toàn vẹn dữ liệu, như ràng buộc khóa chính, khóa ngoại, hoặc các ràng buộc không null. Ví dụ**:** Nếu một ràng buộc yêu cầu rằng số dư của tất cả các tài khoản trong ngân hàng phải lớn hơn hoặc bằng 0, tính nhất quán đảm bảo rằng sau mỗi transaction, ràng buộc này vẫn được duy trì.
- **Isolation (Tính Cách Ly)**: Tính cách ly đảm bảo rằng các transaction đồng thời không ảnh hưởng lẫn nhau. Các thay đổi được thực hiện bởi một transaction chưa hoàn tất sẽ không hiển thị cho các giao dịch khác. Ví dụ**:** Nếu hai transaction cùng cập nhật số dư của một tài khoản, tính cách ly đảm bảo rằng mỗi transaction sẽ không thấy các thay đổi của transaction kia cho đến khi cả hai đều hoàn tất.
- **Durability (Tính Bền Vững)**: Tính bền vững đảm bảo rằng một khi transaction đã được commit, các thay đổi của nó sẽ được lưu trữ vĩnh viễn, ngay cả khi hệ thống gặp sự cố ngay sau đó. Ví dụ**:** Khi một giao dịch cập nhật số dư tài khoản và cam kết thành công, thay đổi này sẽ tồn tại ngay cả khi hệ thống gặp sự cố sau đó.

Các mức cách ly (Isolation level)

Mức cách ly xác định các quy tắc cho những thay đổi nào là hiển thị và không hiển thị bên trong và bên ngoài một transaction. Điều này là quan trọng, vì nó có thể ảnh hưởng đến tinh chính xác và cả hiệu suất của một transaction. Dưới đây là bốn mức cách ly trong chuẩn ANSI SQL:

READ UNCOMMITTED

READ UNCOMMITTED là mức cách ly thấp nhất trong các mức cách ly của cơ sở dữ liệu. Ở mức này, các transaction có thể đọc các thay đổi chưa được commit từ các transaction khác. Điều này có nghĩa là một transaction có thể thấy dữ liệu bị thay đổi bởi một transaction khác, ngay cả khi transaction đó chưa hoàn thành và commit.

Đặc Điểm:

- **Dirty Read (Đọc dữ liệu "bẩn")**: Transaction có thể đọc các thay đổi chưa được commit từ các transaction khác. Điều này có thể dẫn đến việc đọc dữ liệu không nhất quán.
- **Không có bảo đảm về tính nhất quán**: Vì các thay đổi chưa được commit có thể bị rollback, dữ liệu đọc được ở mức cách ly này có thể không chính xác hoặc không đáng tin cậy.
- **Hiệu suất cao**: Mức cách ly này có thể cải thiện hiệu suất vì nó ít tốn kém về mặt tài nguyên và khóa dữ liệu so với các mức cách ly cao hơn.

Ví dụ

Giả sử bạn có hai transaction đang chạy đồng thời:

Transaction 1

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
-- Transaction 1 chưa COMMIT
```

Transaction 2

```
START TRANSACTION;
```

```
SELECT balance FROM accounts WHERE account_id = 1;
-- Transaction 2 đọc được dữ liệu chưa được commit từ Transaction 1
COMMIT;
```

Trong ví dụ này, transaction 2 có thể đọc số dư tài khoản đã bị giảm 100 mặc dù transaction 1 chưa được commit. Nếu sau đó transaction 1 bị rollback, transaction 2 đã đọc dữ liệu không chính xác.

READ COMMITTED

READ COMMITTED là mức cách ly phổ biến và được sử dụng rộng rãi trong nhiều hệ quản trị cơ sở dữ liệu. Ở mức cách ly này, một transaction chỉ có thể đọc các thay đổi đã được commit bởi các transaction khác. Điều này giúp tránh được các vấn đề liên quan đến dirty read (đọc dữ liệu "bẩn"), đảm bảo rằng các dữ liệu đọc được luôn nhất quán và đáng tin cậy.

Đặc Điểm:

- **Không có dirty read**: Transaction chỉ có thể đọc các dữ liệu đã được commit, do đó tránh được việc đọc dữ liệu không nhất quán.
- **Non-repeatable read**: Mặc dù các dirty read được ngăn chặn, nhưng có thể xảy ra hiện tượng non-repeatable read, nghĩa là dữ liệu có thể thay đổi giữa các lần đọc trong cùng một giao dịch nếu một transaction khác commit thay đổi dữ liệu đó.
- **Hiệu suất tốt**: Cân bằng giữa hiệu suất và tính nhất quán, phù hợp cho nhiều ứng dụng.

Ví dụ

Giả sử bạn có hai transaction đang chạy đồng thời:

Transaction 1

```
START TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
-- Transaction 1 chưa COMMIT
```

Transaction 2

```
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1;
COMMIT;
```

Trong ví dụ này, Transaction 2 sẽ không thấy số dư tài khoản đã bị giảm 100 bởi Transaction 1 vì Transaction 1 chưa được commit. Transaction 2 sẽ chỉ đọc dữ liệu đã được commit trước đó.

REPEATABLE READ

REPEATABLE READ là mức cách ly mặc định trong MySQL và được coi là một trong những mức cách ly mạnh mẽ nhất. Ở mức cách ly này, một transaction sẽ thấy cùng một tập hợp dữ liệu nếu nó đọc cùng một truy vấn nhiều lần trong suốt quá trình thực thi, ngay cả khi các transaction khác đã thay đổi dữ liệu đó và commit các thay đổi của chúng. **REPEATABLE READ** ngăn chặn cả dirty read và non-repeatable read.

Đặc Điểm:

- **Ngăn chặn dirty read**: Giao dịch chỉ có thể đọc dữ liệu đã được commit.
- **Ngăn chặn non-repeatable read**: Dữ liệu đọc được trong transaction sẽ không thay đổi cho đến khi giao dịch kết thúc, ngay cả khi các giao dịch khác đã cam kết thay đổi.
- **Phantom Read**: Vẫn có thể xảy ra hiện tượng phantom read, nghĩa là các row mới có thể xuất hiện hoặc các row hiện có có thể biến mất nếu một transaction khác chèn hoặc xóa row trong phạm vi đã được đọc.
- **Sử dụng MVCC (Multiversion Concurrency Control)**: MySQL sử dụng MVCC để đảm bảo các transaction đọc có thể thấy trạng thái nhất quán của dữ liệu.
- Hiệu suất sẽ giảm hơn so với hai mức cách ly bên trên.

Ví Dụ

Giả sử bạn có hai transaction đang chạy đồng thời:

Transaction 1

```
START TRANSACTION;
```

```
SELECT balance FROM accounts WHERE account_id = 1;
-- Transaction 1 đọc số dư tài khoản lần đầu
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
-- Transaction 1 chưa COMMIT
```

Transaction 2

```
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1;
-- Transaction 2 đọc số dư tài khoản, thấy số dư trước khi Transaction 1 cập nhật
COMMIT;
```

Trong ví dụ này, Transaction 2 sẽ thấy số dư ban đầu của tài khoản trước khi Transaction 1 cập nhật. Ngay cả khi Transaction 1 commit thay đổi, Transaction 2 sẽ không thấy thay đổi đó nếu nó thực hiện lại truy vấn.

SERIALIZABLE

SERIALIZABLE là mức cách ly cao nhất trong các mức cách ly của cơ sở dữ liệu. Ở mức cách ly này, mỗi giao dịch được thực hiện một cách tuần tự, không có transaction nào có thể nhìn thấy các thay đổi của giao dịch khác cho đến khi transaction đó hoàn thành. Điều này đảm bảo rằng không có phantom read, non-repeatable read hay dirty read.

Đặc Điểm:

- **Không có dirty read**: Giao dịch chỉ đọc các thay đổi đã được commit.
- **Không có non-repeatable read**: Dữ liệu đọc được sẽ không thay đổi trong suốt thời gian transaction.
- **Không có phantom read**: Transaction sẽ không thấy dữ liệu mới được thêm vào bởi các transaction khác.
- **Khóa toàn bộ các hàng dữ liệu**: Mỗi transaction sẽ khóa các dãy dữ liệu nó truy cập, ngăn chặn các transaction khác thay đổi dữ liệu trong các hàng đó.

- Hiệu suất của SERIALIZABLE sẽ là thấp nhất trong bốn loại cách ly được đề cập ở đây.

Ví dụ

Giả sử bạn có hai transaction đang chạy đồng thời:

Transaction 1

```
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1 FOR UPDATE;
-- Transaction 1 khóa tài khoản 1 để tránh các thay đổi từ các giao dịch khác
UPDATE accounts SET balance = balance - 100 WHERE account_id = 1;
COMMIT;
```

Transaction 2

```
START TRANSACTION;
SELECT balance FROM accounts WHERE account_id = 1;
-- Transaction 2 sẽ bị chặn lại cho đến khi Transaction 1 hoàn thành
COMMIT;
```

Trong ví dụ này, Transaction 2 sẽ bị chặn lại cho đến khi Transaction 1 hoàn thành và commit. Điều này đảm bảo rằng Transaction 2 sẽ không thấy bất kỳ thay đổi nào từ Transaction 1 cho đến khi Transaction 1 hoàn tất.

Kết Luận

Các transaction trong MySQL với các tính chất ACID giúp đảm bảo tính toàn vẹn dữ liệu và xử lý các lỗi có thể xảy ra trong quá trình thực thi. Việc chọn mức cách ly phù hợp tùy thuộc vào yêu cầu cụ thể của ứng dụng và mức độ cân bằng giữa hiệu suất và độ an toàn dữ liệu.

Deadlocks

Deadlocks là gì ?, xảy ra như thế nào

Deadlocks (Khoá chết) xảy ra khi hai hoặc nhiều transaction đồng thời giữ các khóa trên một tập hợp tài nguyên và mỗi transaction chờ đợi một khóa đang được giữ bởi một transaction khác trong tập hợp đó (Transaction A đợi Transaction B, nhưng B cũng lại đợi A). Điều này tạo ra một vòng lặp phụ thuộc mà không thể giải quyết được, và tất cả các transaction liên quan sẽ chờ đợi vô thời hạn trừ khi có biện pháp can thiệp để giải quyết tình trạng này.. Ví dụ, xem xét hai transaction sau đây đang chạy trên bảng `StockPrice`:

Transaction 1:

```
START TRANSACTION;
UPDATE StockPrice SET close = 45.50 WHERE stock_id = 4 AND date = '2020-05-01';
UPDATE StockPrice SET close = 19.80 WHERE stock_id = 3 AND date = '2020-05-02';
COMMIT;
```

Transaction 2:

```
START TRANSACTION;
UPDATE StockPrice SET high = 20.12 WHERE stock_id = 3 AND date = '2020-05-02';
UPDATE StockPrice SET high = 47.20 WHERE stock_id = 4 AND date = '2020-05-01';
COMMIT;
```

Trong ví dụ này, Transaction 1 và Transaction 2 sẽ tạo ra deadlock như sau:

1. Transaction 1 khóa hàng có stock_id = 4 trước.
2. Transaction 2 khóa hàng có stock_id = 3 trước.
3. Transaction 1 cố gắng khóa hàng có stock_id = 3 nhưng phải chờ vì hàng này đã bị Transaction 2 khóa.
4. Transaction 2 cố gắng khóa hàng có stock_id = 4 nhưng phải chờ vì hàng này đã bị Transaction 1 khóa.

Kết quả là cả hai transaction sẽ chờ đợi vô thời hạn, gây ra deadlock.

Các phương pháp xử lý Deadlocks

Để giải quyết vấn đề này, các hệ thống cơ sở dữ liệu triển khai các dạng phát hiện và thời gian chờ deadlock khác nhau.Ví dụ như storage engine nổi tiếng của MySQL là InnoDB: Nó có khả năng phát hiện các dấu hiệu phụ thuộc vòng tròn, và báo lỗi ngay lập tức. Điều này là rất quan trọng, bởi nếu Deadlocks không được xử lý kịp thời sẽ khiến cả hệ thống chạy rất chậm. Những cơ sở dữ liệu khác khác sẽ tiến hành rollback transaction sau khi truy vấn vượt quá thời gian chờ khóa, điều này không phải lúc nào cũng tốt. Cách mà InnoDB hiện tại xử lý deadlock là rollback giao dịch khóa ít row nhất (một thước đo gần đúng cho cái nào sẽ dễ rollback nhất).

Xử lý Deadlocks trong ứng dụng

Một khi deadlocks xảy ra, chúng không thể bị phá vỡ mà không rollback một hoặc toàn bộ các transaction. Deadlock rất dễ xảy ra trong thực tế, và ứng dụng của bạn nên được thiết kế để xử lý chúng. Cách đơn giản nhất là bạn có thể ngừng transaction và tiến hành retry lại truy vấn của mình.

Ví dụ xử lý Deadlocks

```
try {
    $pdo->beginTransaction();
    // Thực hiện các câu lệnh cập nhật
    $pdo->commit();
} catch (PDOException $e) {
    if ($pdo->inTransaction()) {
        $pdo->rollBack();
    }
    if ($e->getCode() == '40001') { // Mã lỗi deadlock trong MySQL
        // Thử lại giao dịch
    } else {
        throw $e;
```

```
    }
}
```

Kết Luận

Deadlocks là một phần không thể tránh khỏi trong các hệ thống cơ sở dữ liệu sử dụng transaction. Hiểu rõ cách deadlocks xảy ra và cách xử lý chúng sẽ giúp bạn thiết kế các ứng dụng bền vững và đáng tin cậy hơn. Việc triển khai các phương pháp phát hiện và xử lý deadlocks hiệu quả là rất quan trọng để đảm bảo rằng hệ thống của bạn hoạt động mượt mà và không bị gián đoạn.

Transaction trong MySQL

Hiện nay InnoDB là storage engine hỗ trợ transaction mặc định và được khuyến nghị sử dụng nhiều nhất trong MySQL. Hãy cùng mình đi qua một số ý chính về Transaction trong MySQL nhé:

Hiểu về AUTOCOMMIT

Mặc định, một câu lệnh đơn lẻ `INSERT`, `UPDATE`, hoặc `DELETE` được tự động bao bọc trong một transaction và được commit ngay lập tức. Điều này được gọi là chế độ **AUTOCOMMIT**. Bạn có thể bật hoặc tắt biến AUTOCOMMIT cho kết nối hiện tại bằng cách sử dụng lệnh `SET`. Các giá trị 1 và `ON` là tương đương nhau, cũng như 0 và `OFF`. Khi bạn chạy với `AUTOCOMMIT=0`, bạn luôn ở trong một transaction cho đến khi bạn thực thi lệnh `COMMIT` hoặc `ROLLBACK`; sau đó MySQL sẽ bắt đầu một transaction mới ngay lập tức. Với **AUTOCOMMIT** được bật, bạn có thể bắt đầu một transaction nhiều câu lệnh bằng cách sử dụng từ khóa `BEGIN` hoặc `START TRANSACTION`.

MySQL cho phép bạn đặt mức cách ly bằng lệnh `SET TRANSACTION ISOLATION LEVEL`, lệnh này có hiệu lực khi transaction tiếp theo bắt đầu. Bạn có thể đặt mức cách ly cho toàn bộ server trong config của MySQL hoặc chỉ cho session của client hiện tại:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

Tốt nhất là bạn nên đặt mức cách ly bạn sử dụng nhiều nhất ở mức server và chỉ thay đổi nó trong các trường hợp cụ thể. MySQL sử dụng tất cả bốn mức cách ly chuẩn ANSI như đã mô tả ở trên và InnoDB hỗ trợ tất cả chúng.

Ví dụ sử dụng transaction trong MySQL

Bắt Đầu Transaction và tắt **AUTOCOMMIT**

```
-- Tắt AUTOCOMMIT để bắt đầu một transaction
SET AUTOCOMMIT = 0;
```

```
-- Bắt đầu transaction
START TRANSACTION;

-- Thực hiện các câu lệnh trong transaction
UPDATE account SET balance = balance - 100 WHERE account_id = 1;
UPDATE account SET balance = balance + 100 WHERE account_id = 2;

-- Commit transaction
COMMIT;
-- Bật lại AUTOCOMMIT nếu cần
SET AUTOCOMMIT = 1;
```

Cài đặt mức cách ly

```
-- Đặt mức cách ly cho session hiện tại
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
-- Bắt đầu transaction với mức cách ly đã đặt
START TRANSACTION;
-- Thực hiện các câu lệnh trong transaction
UPDATE orders SET status = 'completed' WHERE status = 'pending';
-- Commit transaction
COMMIT;
```

Kết Luận

MySQL, đặc biệt với storage engine InnoDB, cung cấp các cơ chế mạnh mẽ cho transaction để đảm bảo tính nhất quán và an toàn dữ liệu. Hiểu rõ cách sử dụng Transaction cũng như các mức cách ly (Isolation Level) giúp bạn tối ưu được hiệu suất truy vấn, cũng như đảm bảo tính chính xác dữ liệu cho ứng dụng của mình.

Thiết kế và quản lý schema

Section này cung cấp một cái nhìn chi tiết về thiết kế và quản lý schema trong MySQL. Ở đây, mình sẽ cung cấp một số “best practices” xoay quanh việc chọn kiểu dữ liệu tối ưu, quản lý schema, và các lỗi thường gặp trong thiết kế schema.

Chọn loại dữ liệu tối ưu

MySQL hỗ trợ rất nhiều kiểu dữ liệu khác nhau. Việc chọn loại dữ liệu thích hợp cho các cột trong bảng là bước quan trọng đầu tiên trong thiết kế schema cũng như đảm bảo performance cao cho truy vấn. Dưới đây là một số nguyên tắc bạn nên áp dụng, bất kể là bạn đang sử dụng kiểu dữ liệu gì trong bảng của mình:

Nhỏ hơn thì thường là tốt hơn

Sử dụng kiểu dữ liệu nhỏ nhất có thể: Các kiểu dữ liệu nhỏ hơn thường nhanh hơn và tiết kiệm tài nguyên hơn.

Ví dụ: Nếu bạn chỉ cần lưu trữ các giá trị từ 0 đến 255, hãy sử dụng `TINYINT` thay vì `INT`. Điều này không chỉ tiết kiệm không gian lưu trữ mà còn tăng hiệu suất truy vấn.

Tuy nhiên, đừng vì thế mà bạn đánh giá thấp phạm vi giá trị cần lưu trữ: Mặc dù việc sử dụng kiểu dữ liệu nhỏ hơn là tốt, nhưng bạn cần đảm bảo rằng phạm vi giá trị của kiểu dữ liệu đủ lớn để lưu trữ tất cả các giá trị có thể.

Ví dụ: Nếu bạn không chắc chắn liệu số lượng bản ghi của một bảng có thể vượt quá 65,535 (giới hạn của `SMALLINT`), bạn nên sử dụng `INT` để tránh phải thay đổi kiểu dữ liệu sau này, điều này có thể rất tốn thời gian và công sức.

Chọn kiểu dữ liệu nhỏ nhất mà bạn nghĩ rằng sẽ không vượt quá: Nếu bạn không chắc chắn về kiểu dữ liệu nào là tốt nhất, hãy chọn kiểu dữ liệu nhỏ nhất mà bạn nghĩ rằng bạn sẽ không vượt quá.

Ví dụ: Nếu bạn đang thiết kế một bảng để lưu trữ số lượng hàng hóa trong kho, và bạn không chắc chắn liệu số lượng hàng hóa có thể vượt quá 32,767 hay không (giới hạn của `SMALLINT` có dấu), bạn nên chọn `INT` để đảm bảo rằng bạn sẽ không gặp vấn đề với việc vượt quá phạm vi giá trị.

Ví dụ cụ thể:

Chọn kiểu dữ liệu cho một bảng sản phẩm:

```
CREATE TABLE products (
  product_id INT AUTO_INCREMENT PRIMARY KEY,
  product_name VARCHAR(100) NOT NULL,
  price DECIMAL(10, 2) NOT NULL,
  quantity SMALLINT UNSIGNED NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

`product_id` là một khóa chính, nên sử dụng `INT` với `AUTO_INCREMENT`.

`product_name` có thể chứa tới 100 ký tự.

`price` sử dụng `DECIMAL(10, 2)` để lưu trữ giá trị tiền tệ với độ chính xác cao.

`quantity` sử dụng `SMALLINT UNSIGNED` vì số lượng hàng hóa sẽ không âm và ít có khả năng vượt quá 65,535.

`created_at` sử dụng `TIMESTAMP` để tự động lưu trữ thời gian tạo bản ghi.

Tóm tắt lưu ý khi chọn kiểu dữ liệu:

- **Hiểu rõ yêu cầu dữ liệu**: Hãy xem xét kỹ lưỡng các yêu cầu và đặc điểm dữ liệu của bạn để chọn kiểu dữ liệu phù hợp.
- **Cân nhắc tính linh hoạt và khả năng mở rộng**: Hãy đảm bảo rằng kiểu dữ liệu bạn chọn có thể đáp ứng nhu cầu hiện tại và tương lai mà không cần thay đổi lớn.

Càng đơn giản càng tốt

Các kiểu dữ liệu đơn giản, ví dụ dạng INT thường tốn ít CPU hơn để tính toán và xử lý. Trong hầu hết các trường hợp, việc, so sánh các số nguyên thường sẽ nhanh hơn so với so sánh các ký tự vì các bộ ký tự và các quy tắc sắp xếp (collations) làm cho việc so sánh ký tự trở nên phức tạp.

Ví dụ về việc sử dụng các kiểu dữ liệu đơn giản:

Sử dụng các kiểu dữ liệu ngày và giờ có sẵn của MySQL thay vì chuỗi (string): Lưu trữ ngày và giờ dưới dạng các kiểu dữ liệu có sẵn như `DATE`, `TIME`, `DATETIME`, và `TIMESTAMP` thay vì chuỗi sẽ tiết kiệm bộ nhớ và tăng hiệu suất truy vấn.
Ví dụ:

```
CREATE TABLE events (
  event_id INT AUTO_INCREMENT PRIMARY KEY,
  event_name VARCHAR(100) NOT NULL,
  event_date DATE NOT NULL
);
```

Trong ví dụ này, `event_date` được lưu trữ dưới dạng `DATE` thay vì chuỗi. Điều này giúp MySQL xử lý các phép toán liên quan đến ngày nhanh hơn, chẳng hạn như so sánh ngày hoặc tính toán khoảng cách thời gian.

Sử dụng số nguyên thay vì chuỗi (string) cho các giá trị danh mục: Nếu bạn có các danh mục hoặc các giá trị trạng thái, hãy sử dụng các kiểu dữ liệu số nguyên hoặc ENUM thay vì chuỗi.
Ví dụ:

```
CREATE TABLE orders (
  order_id INT AUTO_INCREMENT PRIMARY KEY,
  customer_id INT NOT NULL,
  status ENUM('pending', 'shipped', 'delivered', 'canceled') NOT NULL
);
```

Trong ví dụ này, `status` được lưu trữ dưới dạng ENUM thay vì chuỗi, giúp tiết kiệm không gian và tăng tốc độ truy vấn khi tìm kiếm, so sánh theo trạng thái đơn hàng.

Lợi ích của việc sử dụng kiểu dữ liệu đơn giản:

- **Hiệu suất cao hơn**: Các kiểu dữ liệu đơn giản sẽ tốn ít CPU hơn để xử lý các phép toán, giúp cải thiện hiệu suất tổng thể của hệ thống.
- **Tiết kiệm không gian lưu trữ**: Các kiểu dữ liệu nhỏ hơn chiếm ít không gian lưu trữ hơn, giúp giảm kích thước bảng và tăng tốc độ truy vấn.
- **Tăng tính nhất quán**: Sử dụng các kiểu dữ liệu chuẩn như `DATE`, `TIME`, `INT`, và `ENUM` giúp tăng tính nhất quán trong cơ sở dữ liệu, giảm thiểu lỗi và dễ dàng quản lý hơn.

Tránh NULL nếu có thể

Trong thực tế, mình gặp rất nhiều các schema bao gồm các cột có thể NULL (Nullable) ngay cả khi những cột đó sẽ không bao giờ được để trống giá trị. Thông thường, tốt nhất là chỉ định các cột là NOT NULL trừ khi bạn thực sư có dự định lưu trữ NULL trong chúng.

Tại sao nên tránh sử dụng cột có thể NULL ?

- **Tối ưu hóa truy vấn khó khăn hơn**: MySQL gặp khó khăn hơn trong việc tối ưu hóa các truy vấn liên quan đến các cột NULL-able vì chúng làm cho việc Index và so sánh giá trị trở nên phức tạp hơn. (Hiểu đơn giản, đang có một đám Index dạng số nguyên, tự nhiên lòi ra một ông NULL, biết nhét ông đó vào đâu để tiện truy vấn, so sánh, sắp xếp bây giờ …)
- **Tiêu tốn không gian lưu trữ**: Cột có thể NULL sử dụng nhiều không gian lưu trữ hơn và yêu cầu xử lý đặc biệt bên trong MySQL. (Có thể hiểu đơn giản, MySQL cần lưu trữ thêm một cờ (flag) tại từng cột để đánh dấu cột đó là NULL-able)
- **So sánh giá trị phức tạp hơn**: Các so sánh giá trị với cột có thể NULL phức tạp hơn vì MySQL phải xử lý các giá trị NULL một cách đặc biệt. (tương tự như hai mục trên đã mô tả).

Ví dụ: Trong một bảng người dùng, cột `username` và `password` nên được đặt là NOT NULL vì mỗi người dùng luôn cần có tên đăng nhập và mật khẩu.

```
CREATE TABLE users (
  user_id INT AUTO_INCREMENT PRIMARY KEY,
  username VARCHAR(50) NOT NULL,
  password VARCHAR(100) NOT NULL,
  email VARCHAR(100),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

Các loại số trong MySQL

Có hai loại số: số nguyên (whole numbers) và số thực (real numbers, tức là số có phần thập phân). Nếu bạn lưu trữ số nguyên, hãy sử dụng một trong các kiểu số nguyên sau: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, hoặc `BIGINT`. Các kiểu dữ liệu này yêu cầu không gian lưu trữ lần lượt là 8, 16, 24, 32, và 64 bit. Chúng có thể lưu trữ các giá trị từ $-2^{(n-1)}$ đến $2^{(N-1)} - 1$, trong đó N là số bit không gian lưu trữ mà chúng sử dụng. Các kiểu số nguyên có thể sử dụng thuộc tính `UNSIGNED`, khi đó nó sẽ không cho phép chúng lưu giá trị âm và tăng gấp đôi giới hạn trên của các giá trị dương mà chúng có thể lưu trữ.

Ví dụ: Một `TINYINT UNSIGNED` có thể lưu trữ các giá trị từ 0 đến 255 thay vì từ −128 đến 127.

Số nguyên

TINYINT

- **Kích thước lưu trữ**: 1 byte
- **Phạm vi giá trị có dấu**: -128 đến 127
- **Phạm vi giá trị không dấu**: 0 đến 255

`TINYINT` được sử dụng khi bạn cần lưu trữ các giá trị số nhỏ, chẳng hạn như cờ trạng thái (flag), mã trạng thái hoặc giá trị boolean (0 hoặc 1).

SMALLINT

- **Kích thước lưu trữ**: 2 byte
- **Phạm vi giá trị có dấu**: -32,768 đến 32,767
- **Phạm vi giá trị không dấu**: 0 đến 65,535

`SMALLINT` phù hợp để lưu trữ các giá trị số lớn hơn `TINYINT` nhưng vẫn nhỏ hơn `INT`, chẳng hạn như các mã định danh (ID) hoặc số lượng nhỏ.

MEDIUMINT

- **Kích thước lưu trữ**: 3 byte
- **Phạm vi giá trị có dấu**: -8,388,608 đến 8,388,607
- **Phạm vi giá trị không dấu**: 0 đến 16,777,215

`MEDIUMINT` được sử dụng khi bạn cần lưu trữ các giá trị số lớn hơn `SMALLINT` nhưng không cần tới phạm vi của `INT`.

INT (INTEGER)

- **Kích thước lưu trữ**: 4 byte
- **Phạm vi giá trị có dấu**: -2,147,483,648 đến 2,147,483,647
- **Phạm vi giá trị không dấu**: 0 đến 4,294,967,295

BIGINT

- **Kích thước lưu trữ**: 8 byte
- **Phạm vi giá trị có dấu**: -9,223,372,036,854,775,808 đến 9,223,372,036,854,775,807
- **Phạm vi giá trị không dấu**: 0 đến 18,446,744,073,709,551,615

`BIGINT` được sử dụng khi bạn cần lưu trữ các giá trị số rất lớn vượt quá phạm vi của `INT`, chẳng hạn như số liệu thống kê hoặc dữ liệu tài chính lớn.

Độ rộng của kiểu số nguyên

MySQL cho phép bạn chỉ định "độ rộng" cho các kiểu số nguyên, chẳng hạn như `INT(11)`. Điều này không có ý nghĩa đối với hầu hết các ứng dụng: nó không giới hạn phạm vi giá trị hợp lệ mà chỉ đơn

giản là chỉ định số ký tự mà các công cụ tương tác của MySQL (chẳng hạn như công cụ command line) sẽ sử dụng cho mục đích hiển thị. Về mặt lưu trữ và tính toán, `INT(1)` giống hệt `INT(20)`.

Ví dụ

Tạo bảng với các kiểu số nguyên khác nhau:

```
CREATE TABLE number_examples (
    tiny_int_col TINYINT,
    small_int_col SMALLINT,
    medium_int_col MEDIUMINT,
    int_col INT,
    big_int_col BIGINT,
    unsigned_tiny_int_col TINYINT UNSIGNED,
    unsigned_small_int_col SMALLINT UNSIGNED,
    unsigned_medium_int_col MEDIUMINT UNSIGNED,
    unsigned_int_col INT UNSIGNED,
    unsigned_big_int_col BIGINT UNSIGNED
);
```

Kết luận

- **Chọn kiểu dữ liệu nhỏ nhất**: Sử dụng kiểu dữ liệu nhỏ nhất có thể lưu trữ giá trị của bạn.
- **Xem xét phạm vi giá trị**: Đảm bảo phạm vi giá trị phù hợp với nhu cầu của bạn để tránh phải thay đổi sau này.
- **UNSIGNED**: Sử dụng thuộc tính `UNSIGNED` nếu không cần lưu trữ giá trị âm để tăng gấp đôi giới hạn trên của các giá trị dương.
- **Độ rộng kiểu số nguyên**: Độ rộng kiểu số nguyên chỉ ảnh hưởng đến hiển thị, không ảnh hưởng đến lưu trữ hay tính toán.

Số thực

Số thực là các số có phần thập phân. Tuy nhiên, chúng không chỉ dành cho các số thập phân: bạn cũng có thể sử dụng `DECIMAL` để lưu trữ các số nguyên lớn đến mức chúng không thể nằm gọn trong `BIGINT`. MySQL hỗ trợ cả các kiểu số chính xác và không chính xác.

Các kiểu dữ liệu số thực

Kiểu FLOAT và DOUBLE

- **FLOAT**: Sử dụng toán học số thực chuẩn để thực hiện các tính toán xấp xỉ. Cột `FLOAT` sử dụng 4 byte bộ nhớ.
- **DOUBLE**: Sử dụng 8 byte bộ nhớ và có độ chính xác cao hơn và phạm vi giá trị lớn hơn so với `FLOAT`.

Kiểu DECIMAL

Kiểu `DECIMAL` hỗ trợ các tính toán chính xác cho các số thập phân. Kiểu dữ liệu này nên được sử dụng khi bạn cần kết quả chính xác cho các số thập phân, chẳng hạn như khi lưu trữ dữ liệu tài chính.

Ví dụ:

```
CREATE TABLE financial_data (
    amount DECIMAL(10, 4)
);
```

Lựa chọn kiểu dữ liệu số thực tối ưu

- **FLOAT và DOUBLE**: Các kiểu dữ liệu số thực thường sử dụng ít không gian hơn so với `DECIMAL` để lưu trữ cùng một phạm vi giá trị. `FLOAT` sử dụng 4 byte bộ nhớ, trong khi `DOUBLE` sử dụng 8 byte và có độ chính xác và phạm vi giá trị lớn hơn `FLOAT`.

- **DECIMAL**: Bạn nên sử dụng `DECIMAL` khi cần kết quả chính xác cho các số thập phân, ví dụ như khi lưu trữ dữ liệu tài chính. Tuy nhiên, vì yêu cầu không gian lưu trữ và chi phí tính toán lớn hơn, bạn chỉ nên sử dụng `DECIMAL` khi thực sự cần thiết.

Sử dụng BIGINT thay cho DECIMAL trong một số trường hợp

Trường hợp phổ biến khi bạn xử lý các đơn vị tiền tệ số lẻ (ví dụ như USD hoặc các loại tiền kỹ thuật số như Bitcoin), thì bạn nên sử dụng `BIGINT` thay vì `DECIMAL` và lưu trữ dữ liệu dưới dạng một bội số đơn vị nhỏ nhất của tiền tệ mà bạn cần xử lý. Giả sử bạn cần lưu trữ dữ liệu tài chính về USD, thì bạn có thể nhân tất cả các số tiền lên 100 (Với đơn vị là Cent, vì 1 USD = 100 Cent; Cent cũng là đơn vị nhỏ nhất của USD) và lưu kết quả trong một cột `BIGINT`. Điều này giúp tránh sự không chính xác của lưu trữ số thực và chi phí tính toán chính xác của `DECIMAL`.

Kết Luận

- **FLOAT và DOUBLE**: Sử dụng cho các tính toán xấp xỉ với yêu cầu bộ nhớ ít hơn.
- **DECIMAL**: Sử dụng khi cần kết quả chính xác cho các số thập phân, đặc biệt là dữ liệu tài chính.
- **BIGINT**: Trong các trường hợp khối lượng lớn, lưu trữ dữ liệu tài chính dưới dạng bội số của phần nhỏ nhất của tiền tệ có thể là lựa chọn tối ưu hơn so với `DECIMAL`.

Các kiểu dữ liệu chuỗi (String type) trong MySQL

MySQL hỗ trợ khá nhiều kiểu dữ liệu chuỗi, với nhiều biến thể cho mỗi loại. Mỗi cột chuỗi có thể có bộ ký tự (**charset**) và bộ quy tắc sắp xếp riêng (**collation**). Hai loại kiểu dữ liệu chuỗi chính là `VARCHAR` và `CHAR`: chúng lưu trữ các giá trị ký tự. Dưới đây là một so sánh tổng quát giữa hai loại này.

Kiểu VARCHAR và CHAR

VARCHAR

- **Lưu trữ chuỗi ký tự có độ dài biến đổi**: `VARCHAR` lưu trữ chuỗi ký tự có độ dài biến đổi và là kiểu dữ liệu chuỗi phổ biến nhất. Nó có thể yêu cầu ít không gian lưu trữ hơn so với các kiểu độ dài cố định vì nó chỉ sử dụng không gian cần thiết (tức là ít không gian hơn được sử dụng để lưu trữ các giá trị ngắn hơn).
- **Dung lượng lưu trữ bổ sung**: `VARCHAR` sử dụng 1 hoặc 2 byte bổ sung để ghi lại độ dài của giá trị: 1 byte nếu độ dài tối đa của cột là 255 byte trở xuống, và 2 byte nếu dài hơn.

  Ví dụ: Với bộ ký tự `latin1`, một cột `VARCHAR(10)` sẽ sử dụng tối đa 11 byte lưu trữ. Một cột `VARCHAR(1000)` có thể sử dụng tối đa 1,002 byte, vì nó cần 2 byte để lưu trữ thông tin độ dài.

- **Tối ưu hóa hiệu suất nhờ tiết kiệm không gian**: `VARCHAR` giúp cải thiện hiệu suất vì nó tiết kiệm không gian (do linh hoạt cấp phát bộ nhớ theo độ dài). Tuy nhiên đây cũng là một điểm bất lợi của `VARCHAR`: vì độ dài là biến đổi, nên các thao tác cập nhật dữ liệu sẽ phức tạp hơn và phụ thuộc vào storage engine mà bạn đang sử dụng. Tốt nhất, bạn nên chọn `VARCHAR` cho các cột mà không thường xuyên thay đổi.
- **Sử dụng khi**:
  - Độ dài tối đa của cột lớn hơn nhiều so với độ dài trung bình.
  - Các cập nhật tới trường này hiếm, vì vậy phân mảnh không phải là vấn đề.
  - Bạn đang sử dụng bộ ký tự phức tạp như UTF-8, mỗi ký tự sử dụng số byte lưu trữ biến đổi.

CHAR

- **Lưu trữ chuỗi ký tự có độ dài cố định**: `CHAR` là kiểu độ dài cố định: MySQL luôn phân bổ đủ không gian cho số ký tự được chỉ định cho cột `CHAR`. Khi lưu trữ giá trị kiểu `CHAR`, MySQL sẽ loại bỏ các khoảng trắng ở cuối giá trị. Khi so sánh các giá trị `CHAR`, MySQL sẽ thêm các khoảng trắng cần thiết để đảm bảo các giá trị có độ dài bằng nhau.

  Ví dụ: `CHAR(10)` luôn sử dụng 10 byte lưu trữ bất kể giá trị thực tế có độ dài bao nhiêu.

- **Sử dụng khi**:

    - Bạn muốn lưu trữ các chuỗi rất ngắn hoặc tất cả các giá trị gần như có cùng độ dài.
    - **Ví dụ**: `CHAR` là lựa chọn tốt cho các giá trị MD5 của mật khẩu người dùng, luôn có cùng độ dài.
- `CHAR` **tốt hơn** **`VARCHAR`** **cho dữ liệu thay đổi thường xuyên**: `CHAR` tốt hơn `VARCHAR` cho dữ liệu thay đổi thường xuyên vì hàng có độ dài cố định và không dễ bị phân mảnh.
    - **Ví dụ**: Một cột `CHAR(1)` được thiết kế để chứa các giá trị `Y` và `N` sẽ sử dụng chỉ 1 byte trong bộ ký tự đơn byte, nhưng `VARCHAR(1)` sẽ sử dụng 2 byte vì cần thêm một byte lưu độ dài.

So Sánh **`VARCHAR`** **và** **`CHAR`**

Độ dài lưu trữ:

- `VARCHAR`: Độ dài biến đổi, sử dụng thêm 1 hoặc 2 byte để lưu trữ độ dài.
- `CHAR`: Độ dài cố định, luôn chiếm số byte cố định được chỉ định.

Hiệu suất:

- `VARCHAR`: Tiết kiệm không gian nhưng có thể gây ra phân mảnh khi các hàng thay đổi kích thước.
- `CHAR`: Sử dụng nhiều không gian hơn cho các giá trị ngắn nhưng ít bị phân mảnh hơn.

Trường hợp sử dụng:

- `VARCHAR`: Phù hợp cho các chuỗi có độ dài biến đổi và ít thay đổi.
- `CHAR`: Phù hợp cho các chuỗi ngắn hoặc có độ dài cố định, thay đổi thường xuyên.

Ví dụ

Bảng chứa các cột **`VARCHAR`** **và** **`CHAR`**:

```
CREATE TABLE users (
    username VARCHAR(50), -- tên người dùng có thể có độ dài biến đổi
```

```
    password CHAR(32), -- giá trị MD5 của mật khẩu luôn có độ dài cố định
    email VARCHAR(100) -- địa chỉ email có thể có độ dài biến đổi
);
```

Trong ví dụ này:

- `username` sử dụng `VARCHAR(50)` vì tên người dùng có thể có độ dài khác nhau.
- `password` sử dụng `CHAR(32)` vì giá trị MD5 của mật khẩu luôn có độ dài cố định.
- `email` sử dụng `VARCHAR(100)` vì địa chỉ email có thể có độ dài khác nhau.

Bằng cách hiểu rõ các đặc điểm và tình huống sử dụng của `VARCHAR` và `CHAR`, bạn có thể chọn kiểu dữ liệu string phù hợp nhất cho ứng dụng của mình, từ đó tối ưu hóa hiệu suất và tiết kiệm tài nguyên.

Kiểu dữ liệu BLOB và TEXT

BLOB (Binary Large Object)

Định nghĩa: BLOB là kiểu dữ liệu dùng để lưu trữ các chuỗi nhị phân lớn như hình ảnh, video, hoặc các tệp nhị phân khác.

Các loại BLOB:

- **TINYBLOB**: Lưu trữ dữ liệu lên đến 255 byte.
- **BLOB**: Lưu trữ dữ liệu lên đến 65,535 byte (64 KB).
- **MEDIUMBLOB**: Lưu trữ dữ liệu lên đến 16,777,215 byte (16 MB).
- **LONGBLOB**: Lưu trữ dữ liệu lên đến 4,294,967,295 byte (4 GB).

TEXT

Định nghĩa: TEXT là kiểu dữ liệu dùng để lưu trữ các chuỗi ký tự lớn như văn bản, bài viết, hoặc các đoạn mô tả phức tạp.

Các loại TEXT:

- **TINYTEXT**: Lưu trữ chuỗi ký tự lên đến 255 byte.
- **TEXT**: Lưu trữ chuỗi ký tự lên đến 65,535 byte (64 KB).
- **MEDIUMTEXT**: Lưu trữ chuỗi ký tự lên đến 16,777,215 byte (16 MB).
- **LONGTEXT**: Lưu trữ chuỗi ký tự lên đến 4,294,967,295 byte (4 GB).

Sự khác biệt giữa BLOB và TEXT

- **BLOB**: Lưu trữ dữ liệu nhị phân mà không có collation hoặc bộ ký tự.
- **TEXT**: Có bộ ký tự và collation.

Sắp xếp và Index

- MySQL sắp xếp các cột `BLOB` và `TEXT` khác với các kiểu dữ liệu còn lại: thay vì sắp xếp toàn bộ chiều dài của chuỗi, nó chỉ sắp xếp các byte đầu tiên của chiều dài tối đa `max_sort_length`. Bạn có thể thay đổi cấu hình này trong config `max_sort_length` trên MySQL server nếu cần sắp xếp chỉ dựa trên một số ký tự đầu tiên.
- MySQL không thể đánh index toàn bộ chiều dài của các kiểu dữ liệu này và không thể sử dụng các index cho việc sắp xếp.

Ví dụ sử dụng BLOB và TEXT

Tạo bảng với các cột BLOB và TEXT:

```
CREATE TABLE example_blob_text (
    id INT AUTO_INCREMENT PRIMARY KEY,
    description TEXT,
    data BLOB,
    medium_description MEDIUMTEXT,
    medium_data MEDIUMBLOB
);
```

Trong ví dụ này:

- `description` sử dụng `TEXT` để lưu trữ chuỗi ký tự lớn.

- `data` sử dụng `BLOB` để lưu trữ dữ liệu nhị phân lớn.
- `medium_description` và `medium_data` sử dụng `MEDIUMTEXT` và `MEDIUMBLOB` để lưu trữ lượng dữ liệu lớn hơn `TEXT` và `BLOB`.

`VARCHAR` hay `TEXT`:

Mặc dù nghe có vẻ khá giống nhau, mình vẫn có lưu ý cho bạn để lựa chọn giữa hai kiểu dữ liệu này:

Sử Dụng `VARCHAR`:

- Khi bạn biết chiều dài của chuỗi và nó không quá lớn.
- Khi bạn muốn tối ưu hóa bộ nhớ cho các chuỗi ngắn đến vừa.

Sử Dụng `TEXT`:

- Khi bạn cần lưu trữ chuỗi dài, như nội dung bài viết, mô tả sản phẩm, hoặc bất kỳ dữ liệu văn bản nào lớn hơn `VARCHAR`.
- Khi không biết trước chiều dài của chuỗi.

Bonus: Lưu trữ hình ảnh trong MySQL

Trước đây, không phải là hiếm khi một số ứng dụng chấp nhận lưu trữ hình ảnh dưới dạng dữ liệu `BLOB` trong cơ sở dữ liệu MySQL (hoặc lưu dưới dạng Base64 trong các cột dạng TEXT). Phương pháp này ban đầu khá thuận tiện; tuy nhiên, khi kích thước dữ liệu tăng lên, các hoạt động như thay đổi schema trở nên chậm hơn do kích thước của dữ liệu `BLOB` là rất lớn.

Khuyến nghị: Nếu có thể, đừng lưu trữ dữ liệu như hình ảnh trong cơ sở dữ liệu. Thay vào đó, ghi chúng vào một service lưu trữ riêng biệt và sử dụng MySQL để theo dõi vị trí hoặc tên tệp của hình ảnh.

Kết Luận

- **Chọn đúng kiểu dữ liệu**: Sử dụng `BLOB` cho dữ liệu nhị phân và `TEXT` cho dữ liệu ký tự.

- **Hiểu cách lưu trữ**: Nhận biết rằng các giá trị `BLOB` và `TEXT` được lưu trữ khác nhau so với các kiểu dữ liệu khác và có thể yêu cầu không gian lưu trữ bên ngoài.
- **Tránh lưu trữ dữ liệu lớn trực tiếp**: Đối với dữ liệu như hình ảnh, cân nhắc lưu trữ chúng trong một kho lưu trữ đối tượng riêng biệt và theo dõi vị trí trong cơ sở dữ liệu.

Sử dụng ENUM thay cho kiểu chuỗi

Đôi khi bạn có thể sử dụng cột `ENUM` thay vì các kiểu chuỗi thông thường. Một cột `ENUM` có thể lưu trữ một tập hợp các giá trị chuỗi xác định trước. MySQL lưu trữ chúng rất nhỏ gọn, được nén thành 1 hoặc 2 byte tùy thuộc vào số lượng giá trị trong danh sách. Nó lưu trữ mỗi giá trị nội bộ dưới dạng một số nguyên đại diện cho vị trí của nó trong danh sách được định nghĩa. Dưới đây là một ví dụ:

Giả sử bạn có một bảng lưu trữ trạng thái của đơn hàng:

```
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,
    status ENUM('pending', 'shipped', 'delivered', 'canceled') NOT NULL
);
```

Trong ví dụ này: `status` là một cột `ENUM` có thể chứa một trong các giá trị: `'pending'`, `'shipped'`, `'delivered'`, `'canceled'`. MySQL sẽ lưu trữ các giá trị này rất nhỏ gọn, từ đó truy vấn sẽ có hiệu suất tốt hơn.

Lợi ich của việc sử dụng ENUM

- **Tiết kiệm không gian lưu trữ**: Vì MySQL lưu trữ các giá trị `ENUM` dưới dạng số nguyên nhỏ gọn, nó tiết kiệm không gian lưu trữ so với việc lưu trữ chuỗi ký tự thông thường.
- **Tăng tốc độ so sánh**: So sánh các số nguyên nhanh hơn so với so sánh chuỗi ký tự (Do MySQL lưu trữ giá trị `ENUM` dưới dạng các số nguyên).
- **Ràng buộc tính hợp lệ**: Sử dụng `ENUM` giúp ràng buộc các giá trị hợp lệ cho cột, tránh việc nhập sai giá trị.

Hạn chế của ENUM

- **Khó khăn trong việc thêm/bớt giá trị**: Việc thêm hoặc bớt giá trị trong danh sách `ENUM` yêu cầu thay đổi định nghĩa bảng, điều này có thể tốn thời gian đối với bảng lớn.
- **Giới hạn số lượng giá trị**: `ENUM` chỉ hỗ trợ tối đa 65,535 giá trị (Trong thực tế nếu sử dụng quá 20 đến 30 giá trị trong ENUM bạn cũng nên cân nhắc một cách lưu trữ khác)

Khi nào nên sử dụng ENUM

- **Danh sách giá trị cố định**: Khi bạn có một danh sách giá trị cố định và ít thay đổi.
- **Tiết kiệm không gian và tăng tốc độ truy vấn**: Khi bạn muốn tiết kiệm không gian lưu trữ và tăng tốc độ truy vấn.

Ví dụ khác về ENUM

Lưu trữ giới tính của người dùng:

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    gender ENUM('male', 'female', 'other') NOT NULL
);
```

Trong ví dụ này:

`gender` là một cột `ENUM` có thể chứa một trong các giá trị: `'male'`, `'female'`, `'other'`.

Kết Luận

Sử dụng `ENUM` thay cho các kiểu chuỗi thông thường có thể giúp tối ưu hóa không gian lưu trữ và tăng tốc độ truy vấn. Tuy nhiên, bạn cần cân nhắc kỹ khi cần thay đổi giá trị trong danh sách `ENUM`, đặc biệt với các bảng có số lượng bản ghi lớn.

Kiểu dữ liệu ngày và thời gian trong MySQL

MySQL có nhiều loại dữ liệu cho các giá trị ngày và thời gian khác nhau, chẳng hạn như `YEAR` và `DATE`. Đơn vị thời gian nhỏ nhất mà MySQL có thể lưu trữ là microsecond. Thông thường, việc lưu trữ thời gian trong MySQL khá đơn giản, không có nhiều thứ để bàn đến. Vấn đề lớn nhất bạn sẽ gặp trong MySQL là lưu trữ cả ngày và giờ với kiểu dữ liệu thời gian. MySQL cung cấp hai kiểu dữ liệu rất giống nhau cho mục đích này: `DATETIME` và `TIMESTAMP`. Trong hầu hết các trường hợp, hai kiểu dữ liệu này sẽ hoạt động giống nhau. Tuy nhiên, chúng vẫn có những khác biệt mà bạn cần quan tâm khi chọn kiểu dữ liệu. Hãy cùng xem xét:

DATETIME

- **Phạm vi giá trị**: Kiểu dữ liệu này có thể chứa một phạm vi giá trị lớn, từ năm 1000 đến năm 9999, với độ chính xác là một microsecond.
- **Định dạng lưu trữ**: Nó lưu trữ ngày và giờ được gói gọn trong một số nguyên theo định dạng `YYYYMMDDHHMMSS`, độc lập với múi giờ. Điều này khiến nó sử dụng 8 byte không gian lưu trữ.
- **Định dạng hiển thị**: MySQL hiển thị các giá trị `DATETIME` theo định dạng rõ ràng chẳng hạn như `2008-01-16 22:37:08`. Đây là cách biểu diễn ngày và giờ theo chuẩn ANSI.

TIMESTAMP

- **Phạm vi giá trị**: Do chỉ sử dụng 4 byte bộ nhớ, `TIMESTAMP` có phạm vi giá trị từ năm 1970 đến ngày 19 tháng 1 năm 2038.
- **Định dạng lưu trữ**: Như tên gọi, kiểu `TIMESTAMP` lưu trữ số giây đã trôi qua từ nửa đêm, ngày 1 tháng 1 năm 1970, theo Giờ Quốc tế Greenwich (GMT), tương tự như khái niệm timestamp của Unix. `TIMESTAMP` chỉ sử dụng 4 byte bộ nhớ, vì vậy nó có phạm vi nhỏ hơn nhiều so với `DATETIME`: từ năm 1970 đến ngày 19 tháng 1 năm 2038.
- **Múi giờ (Timezone)**: Giá trị mà `TIMESTAMP` hiển thị phụ thuộc vào múi giờ (timezone). MySQL server, hệ điều hành và kết nối client đều có cài đặt múi giờ. Vì vậy, một giá trị `TIMESTAMP` lưu trữ giá trị 0 sẽ hiển thị là 1970-01-01 07:00:00 theo múi giờ Asia/Ho_Chi_Minh, nhanh hơn 7h so với giờ GMT.

- **Đặc tính đặc biệt**: `TIMESTAMP` có những thuộc tính đặc biệt mà `DATETIME` không có: MySQL sẽ đặt giá trị `TIMESTAMP` thành thời gian hiện tại khi bạn chèn một hàng mà không chỉ định giá trị. MySQL cũng cập nhật giá trị của cột `TIMESTAMP` theo mặc định khi bạn cập nhật hàng trừ khi bạn chỉ định giá trị rõ ràng trong câu lệnh UPDATE.

Lưu trữ ngày và thời gian dưới dạng số nguyên

Cả `DATETIME` và `TIMESTAMP` đều buộc bạn phải xử lý các múi giờ trên server và client. Mặc dù `TIMESTAMP` tiết kiệm không gian hơn `DATETIME` (4 byte so với 8 byte, không tính hỗ trợ phần giây), nhưng nó sẽ gặp vấn đề năm 2038 như mình đề cập ở trên.

Các yếu tố cần cân nhắc

- **Phạm vi thời gian cần hỗ trợ**: Bạn cần hỗ trợ ngày và giờ xa đến mức nào (trước hoặc sau hiện tại)?
- **Không gian lưu trữ**: Không gian lưu trữ quan trọng đến mức nào đối với dữ liệu này?
- **Hỗ trợ phần giây**: Bạn có cần hỗ trợ phần giây không?
- **Xử lý ngày, giờ và múi giờ**: Bạn muốn xử lý ngày, giờ và múi giờ trong MySQL hay xử lý nó trong code ứng dụng ?

Lưu trữ dưới dạng Unix Epoch

Để phòng tránh các hạn chế của MySQL trong việc lưu trữ dữ liệu thời gian, ngày càng nhiều ứng dụng chọn cách lưu trữ ngày và giờ dưới dạng Unix epoch, hay số giây kể từ ngày 1 tháng 1 năm 1970 theo UTC. Với kiểu số nguyên có dấu 32-bit, bạn có thể lưu trữ đến năm 2038. Với kiểu số nguyên không dấu 32-bit, bạn có thể lưu trữ đến năm 2106. Với 64-bit, bạn thậm chí có thể lưu trữ vượt xa hơn. Điều này có thể giải quyết được vấn đề mà `TIMESTAMP` gặp phải.

Ví dụ:

Sử dụng DATETIME và TIMESTAMP:

```
CREATE TABLE events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    event_name VARCHAR(100),
    event_date DATETIME,
    event_timestamp TIMESTAMP
);
```

Sử dụng Unix Epoch:

```
CREATE TABLE unix_events (
    event_id INT AUTO_INCREMENT PRIMARY KEY,
    event_name VARCHAR(100),
    event_epoch INT UNSIGNED
);
```

Kết Luận

- **DATETIME**: Sử dụng khi cần lưu trữ một phạm vi rộng các giá trị ngày và giờ, không phụ thuộc vào múi giờ.
- **TIMESTAMP**: Sử dụng khi cần tiết kiệm không gian lưu trữ và giá trị thời gian phụ thuộc vào múi giờ.
- **Unix Epoch**: Tránh các vấn đề xử lý thời gian của MySQL bằng cách lưu trữ ngày và giờ dưới dạng Unix epoch, phần xử lý dữ liệu hiển thị sẽ được đẩy sang ứng dụng.

Việc chọn kiểu dữ liệu ngày và giờ phụ thuộc vào nhu cầu cụ thể của bạn về phạm vi thời gian, không gian lưu trữ và cách bạn muốn xử lý múi giờ.

Dữ liệu JSON trong MySQL

JSON giờ đây đã trở thành một tiêu chuẩn giao tiếp cực kỳ phổ biến, đặc biệt trong thời đại Microservice phát triển. MySQL có kiểu dữ liệu JSON giúp bạn có thể thao tác trực tiếp với cấu trúc dữ liệu này trên bảng. Nhiều người không thích tạo kiểu dữ liệu JSON trên MySQL, mà thích lưu dữ liệu dưới dạng TEXT, bên trong là JSON String. Cũng có trường phái cho rằng nên tận dụng triệt để kiểu dữ liệu này trên MySQL. Để kết luận rằng cách thức nào là tốt nhất có lẽ phụ thuộc vào mỗi người, nên dưới đây mình sẽ trình bày về cách sử dụng và một chút so sánh về tốc độ truy vấn khi sử dụng kiểu dữ liệu JSON.

Trường hợp sử dụng

Giả sử bạn có một bảng lưu trữ thông tin người dùng với yêu cầu dữ liệu linh hoạt và thay đổi thường xuyên. Bạn có thể chọn giữa lưu trữ dữ liệu dưới dạng các trường riêng biệt hoặc sử dụng kiểu dữ liệu JSON để lưu trữ tất cả trong một cột duy nhất.

Ví dụ

Schema truyền thống:

```
CREATE TABLE users (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100),
    address VARCHAR(100)
);
```

Schema sử dụng JSON Column:

```
CREATE TABLE users_json (
    user_id INT AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100),
```

```
    profile JSON
);
```

Trong schema truyền thống, bạn có thể lưu trữ các trường như `username`, `email` và address dưới dạng các cột riêng biệt. Đối với schema sử dụng JSON Column bạn có thể lưu trữ các thông tin khác của người dùng (như trường address) trong cột `profile` dưới dạng JSON.

Lợi ích của việc sử Dụng JSON

- **Linh hoạt**: Dễ dàng thêm các trường mới vào profile mà không cần thay đổi schema.
- **Đơn giản hóa quản lý Schema**: Tránh việc phải thay đổi schema mỗi khi có trường dữ liệu mới.

Hạn chế của việc sử dụng JSON

1. **Hiệu Suất**: Truy vấn trên các trường riêng biệt thường nhanh hơn so với truy vấn trên cột JSON.
2. **Không Gian Lưu Trữ**: Lưu trữ dữ liệu dưới dạng JSON có thể tốn nhiều không gian hơn so với lưu trữ dưới dạng các trường riêng biệt.

So sánh tốc độ truy vấn và kích thước dữ liệu

Tốc Độ Truy Vấn:

Truy vấn trên schema truyền thống:

```
SELECT username, email FROM users WHERE user_id = 1;
```

Truy vấn trên schema JSON:

```
SELECT username, email, JSON_EXTRACT(profile, '$.address') AS address FROM
users_json WHERE user_id = 1;
```

Truy vấn trên bảng có cấu trúc cố định thường nhanh hơn do không cần phân tích cú pháp JSON.

Kích thước dữ liệu:

Kích thước dữ liệu của schema truyền thống thường nhỏ hơn vì không phải lưu trữ cấu trúc JSON và các ký tự bổ sung.

Kết Luận

- **Schema Truyền Thống**: Phù hợp khi dữ liệu có cấu trúc cố định và không thay đổi nhiều. Tốc độ truy vấn nhanh và tiết kiệm không gian lưu trữ.
- **Schema JSON**: Phù hợp khi dữ liệu linh hoạt và thay đổi thường xuyên. Đơn giản hóa quản lý schema nhưng có thể tốn không gian lưu trữ hơn và truy vấn chậm hơn.

Lựa chọn giữa việc sử dụng kiểu dữ liệu JSON hoặc các trường riêng biệt phụ thuộc vào nhu cầu cụ thể của bạn về tính linh hoạt, hiệu suất và không gian lưu trữ. Bằng cách hiểu rõ các ưu và nhược điểm của từng phương pháp, bạn có thể chọn giải pháp phù hợp nhất cho ứng dụng của mình.

Lựa chọn kiểu dữ liệu tốt cho cột định danh (ID)

Trong cơ sở dữ liệu, định danh là cách bạn tham chiếu đến một hàng và thường là yếu tố làm cho nó trở nên duy nhất (unique). Ví dụ, nếu bạn có một bảng về người dùng, bạn có thể muốn gán cho mỗi người dùng một ID số hoặc một tên đăng nhập duy nhất. Trường này thông thường có thể là một phần hoặc toàn bộ của khoá chính (`PRIMARY KEY`).

Tầm quan trọng của kiểu dữ liệu cho cột định danh

Chọn một kiểu dữ liệu tốt cho cột định danh là rất quan trọng. Bạn sẽ thường xuyên sử dụng cột định danh này trong các phép toán như JOIN, hoặc dùng nó để tìm dữ liệu trong các bảng khác. Bạn cũng có khả năng sử dụng chúng trong các bảng khác như khóa ngoại, vì vậy khi bạn chọn kiểu dữ liệu cho một cột định danh, bạn có thể đang chọn kiểu dữ liệu cho các bảng liên quan nữa.

Các yếu tố cần xem xét khi chọn kiểu dữ liệu cho cột định danh

Kiểu lưu trữ và hiệu suất so sánh:

Khi chọn kiểu dữ liệu cho cột định danh, bạn không những phải cân nhắc cách MySQL lưu trữ, mà còn là cách MySQL xử lý tính toán và so sánh trên các cột này. Ví dụ: MySQL lưu trữ các kiểu `ENUM` và `SET` dưới dạng số nguyên nhưng sẽ chuyển đổi chúng thành chuỗi khi thực hiện các phép so sánh.

Khi bạn xem xét kiểu dữ liệu định danh, hãy chắc chắn rằng bạn sử dụng cùng một kiểu trong tất cả các bảng liên quan (khoá chính - khoá ngoại). Các kiểu dữ liệu cần phải khớp chính xác, bao gồm cả các thuộc tính như `UNSIGNED`. **Ví dụ**: Nếu bạn sử dụng `UNSIGNED INT` cho cột định danh trong một bảng, hãy sử dụng cùng kiểu đó trong các bảng khác mà trường này làm khóa ngoại. Trộn lẫn các kiểu dữ liệu khác nhau có thể gây ra các vấn đề về hiệu suất: đặc biệt trong tình huống xảy ra chuyển đổi kiểu ngầm trong các phép so sánh có thể tạo ra các lỗi khó phát hiện. Ngoài ra, phép JOIN trên các cột khác kiểu dữ liệu cũng sẽ không đạt hiệu suất tốt nhất.

Chọn kích thước nhỏ nhất:

Chọn kích thước nhỏ nhất có thể chứa phạm vi giá trị yêu cầu và sẵn sàng cho sự phát triển trong tương lai nếu cần thiết. **Ví dụ**: Nếu bạn có một cột `province_id` lưu trữ ID các thành phố ở Việt Nam, bạn không cần hàng ngàn hay hàng triệu giá trị, vì vậy bạn không nên sử dụng kiểu dữ liệu `INT`. Kiểu dữ liệu `TINYINT` là vừa đủ cho tình huống này và cả trong tương lai.

Các lời khuyên khi chọn kiểu dữ liệu cho cột định danh

Sử dụng kiểu số nguyên khi có thể:

Ví dụ: Sử dụng `INT` hoặc `BIGINT` cho các cột định danh.

Sử dụng kiểu dữ liệu nhỏ gọn:

Ví dụ: Sử dụng `TINYINT` cho các giá trị có phạm vi nhỏ (ví dụ định danh các bảng ít dữ liệu).

Đồng nhất các thuộc tính:

Ví dụ: Đảm bảo rằng tất cả các cột `id` trong các bảng liên quan đều có cùng thuộc tính, ví dụ như `UNSIGNED`.

Ví dụ

```
CREATE TABLE users (
    user_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(50) NOT NULL,
    email VARCHAR(100) NOT NULL
);

CREATE TABLE orders (
    order_id INT UNSIGNED AUTO_INCREMENT PRIMARY KEY,
    user_id INT UNSIGNED NOT NULL,
    order_date DATETIME NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Trong ví dụ này: `user_id` và `order_id` đều sử dụng `INT UNSIGNED` để đảm bảo đồng nhất kiểu dữ liệu và tối ưu hiệu suất truy vấn khi `JOIN`. Hai giá trị định danh này được thiết kế sử dụng kiểu số nguyên, có độ lớn phù hợp để sẵn sàng cho sự phát triển của ứng dụng trong tương lai.

Những vấn đề trong thiết kế Schema MySQL

Chúng ta vừa đi qua một số “best practices” trong thiết kế Schema trong MySQL. Trong các tình huống thực tế, ngay cả khi bạn đã tuân thủ triệt để các “best practices”, vẫn có khả năng bạn phạm phải một số vấn đề khi thiết kế Schema. Dưới đây là một số vấn đề mình hay gặp:

Quá nhiều cột

Khi bảng có quá nhiều cột, việc truy xuất dữ liệu từ bảng này có thể trở nên chậm chạp. Mỗi lần truy vấn, MySQL phải đọc toàn bộ dữ liệu, kể cả các cột không cần thiết, làm tăng thời gian xử lý và tiêu tốn tài nguyên. Các bảng có nhiều cột tiêu tốn nhiều bộ nhớ hơn, đặc biệt là khi sử dụng buffer pool trong InnoDB. Điều này có thể dẫn đến việc bộ nhớ bị đầy nhanh chóng và ảnh hưởng đến hiệu suất

của toàn bộ hệ thống. Chính vì thế nếu có thể, bạn nên tách một bảng to thành các bảng nhỏ hơn (theo ý kiến cá nhân của mình, bạn nên giữ bảng < 100 cột là phù hợp).

Quá nhiều joins

Những vấn đề bạn sẽ gặp phải khi sử dụng quá nhiều Joins:

- Mỗi phép join đòi hỏi MySQL phải tìm và so khớp các hàng từ các bảng khác nhau, điều này có thể tốn nhiều thời gian và tài nguyên hệ thống.
- Với các câu JOINS phức tạp, MySQL có thể chọn các plan thực thi không tối ưu, dẫn đến thời gian phản hồi chậm.
- Khi MySQL thực thi các truy vấn với nhiều joins, nó có thể cần lưu trữ tạm thời nhiều dữ liệu trong bộ nhớ, dẫn đến quá tải.
- Các joins phức tạp có thể dẫn đến lock bảng, làm giảm hiệu suất của các truy vấn khác đang cố gắng truy cập cùng các bảng này

Lời khuyên của mình là bạn nên tách các câu JOINS phức tạp thành nhiều câu truy vấn đơn giản hơn: chuyển sang WHERE IN chẳng hạn. Trong hầu hết các tình huống, nhiều câu query đơn giản hơn có thể đem lại hiệu suất truy vấn tốt hơn. Ví dụ phần này mình sẽ để ở mục Tối ưu hoá hiệu suất truy vấn bên dưới.

Tránh sử dụng NULL một cách cực đoan

Phía bên trên mình đã mô tả về lợi ích của việc tránh lạm dụng NULL-able. Tuy nhiên, cũng đừng nên tránh sử dụng NULL đến mức cực đoan. Dưới đây là một ví dụ mà mình đã thấy khá thường xuyên:

```
CREATE TABLE ... (
    datetime DATETIME NOT NULL DEFAULT '0000-00-00 00:00:00'
);
```

Giá trị mặc định kia có thể gây ra nhiều vấn đề bên trong codebase của bạn khi parse dữ liệu (Bạn có thể cấu hình SQL_MODE của MySQL để không cho phép các ngày vô nghĩa như trên). Trong trường hợp này, bạn vẫn có thể sử dụng NULL vì nó thực sự cần thiết.

Tối ưu hoá hiệu suất index

Cách hoạt động của Index trong MySQL

Cách dễ nhất để hiểu cách index hoạt động trong MySQL là liên tưởng đến mục lục của một cuốn sách. Để tìm ra một chương ở đâu trong một cuốn sách, bạn tìm trong mục lục, và nó sẽ cho bạn biết số trang nơi chương đó xuất hiện.

Trong MySQL, index hoạt động theo cách tương tự. Nó tìm kiếm kết quả của truy vấn trên index. Nếu có kết quả phù hợp, storage engine sẽ tìm đến hàng (rows) chứa dữ liệu đó và trả ra kết quả cho bạn. Giả sử bạn chạy truy vấn sau:

```
SELECT first_name FROM users WHERE user_id = 5;
```

Cột `user_id` được đánh index, vì vậy MySQL sẽ sử dụng index để tìm các hàng có `user_id` là 5. Nói cách khác, nó thực hiện một truy vấn trên các giá trị trong index và trả về bất kỳ hàng nào chứa kết quả này.

Cấu trúc của index

Một index chứa các giá trị từ một hoặc nhiều cột trong một bảng. Nếu bạn đánh index nhiều hơn một cột, thứ tự cột rất quan trọng vì MySQL chỉ có thể tìm kiếm hiệu quả theo thứ tự từ trái sang phải. Việc tạo index trên hai cột không giống như việc tạo hai index đơn lẻ trên từng cột riêng lẻ.

Ví dụ

Tạo index trên một cột:

```
CREATE INDEX idx_user_id ON user(user_id);
```

Tạo index trên nhiều cột:

```
CREATE INDEX idx_last_first_name ON user(last_name, first_name);
```

Trong ví dụ này, index `idx_last_first_name` sẽ giúp MySQL tìm kiếm nhanh chóng các hàng dựa trên cột `last_name` và sau đó là `first_name`.

- **Index đơn**: Index chỉ trên một cột.
- **Index đa cột**: Index trên nhiều cột, MySQL sử dụng theo thứ tự cột từ trái sang phải để tìm kiếm.

Lợi ích của Index

- **Tăng tốc độ truy vấn**: Index giúp MySQL tìm kiếm và truy xuất dữ liệu nhanh hơn nhiều so với việc quét toàn bộ bảng.
- **Tối ưu hóa hiệu suất**: Index giúp giảm tải hệ thống bằng cách giảm số lượng hàng cần phải đọc và xử lý.

Hạn chế của Index

- **Tốn không gian lưu trữ**: Index cần không gian lưu trữ bổ sung để lưu trữ cấu trúc dữ liệu của chúng.
- **Tăng thời gian Insert và Update**: Việc insert, update, hoặc delete hàng trong bảng có index có thể chậm hơn vì MySQL cũng phải cập nhật index.

Các loại Index chính trong MySQL

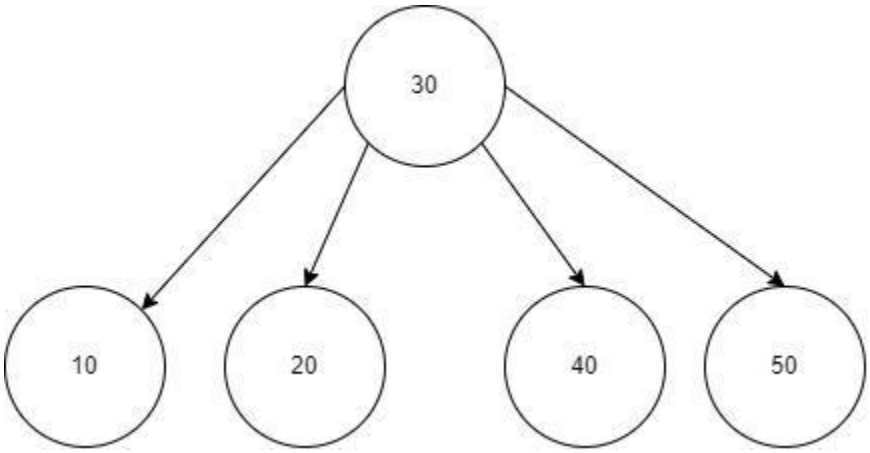
Có nhiều loại index, mỗi loại được thiết kế để hoạt động tốt cho các mục đích khác nhau. Index được triển khai ở tầng storage engine, không phải ở lớp SQL. Do đó, chúng không được chuẩn hóa: việc tạo index có thể khác nhau trong mỗi engine, và không phải tất cả các engine đều hỗ trợ tất cả các loại index. Ngay cả khi nhiều engine hỗ trợ cùng một loại index, chúng cũng có thể triển khai theo các cách khác nhau. Mình sẽ đề cập đến một số loại index chính trong MySQL dưới đây, cùng với đặc điểm và cách sử dụng chúng sao cho đạt hiệu suất tốt nhất:

B-Tree Indexes

B-Tree có thể coi là loại Index phổ biến nhất trong MySQL. Khi nhắc đến index mà không đề cập chi tiết hơn, chúng ta có thể ngầm định đang nhắc đến B-Tree index. B-Tree sử dụng cấu trúc dữ liệu B-tree (Balance Tree) để lưu trữ dữ liệu của nó. Hầu hết các storage engine của MySQL hỗ trợ loại index này.

Nguyên lý cơ bản

Ý tưởng chung của B-tree là tất cả các giá trị được lưu trữ theo thứ tự, và mỗi trang lá (leaf page) cách đều nhau từ gốc (Chính vì thế nó được gọi là cây cân bằng - Balance Tree). Cấu trúc của B-tree là một cây, trong đó mỗi node nội tại có thể có nhiều con, và mỗi node lá nằm ở cùng một mức. Dưới đây là một hình ảnh đơn giản mô tả cấu trúc của B-tree:



Trong ví dụ trên:

- Nút gốc chứa giá trị 30.
- Nút gốc có bốn con chứa các giá trị nhỏ hơn và lớn hơn 30.
- Các nút lá chứa các giá trị 10, 20, 40, 50 tất cả đều nằm ở cùng một mức.

Trong quá trình tìm kiếm, ví dụ, cần tìm giá trị 20, storage engine sẽ biết tìm bên trái của nút gốc (Vì bên trái các nút gốc chứa các giá trị < 30). Cứ tiếp tục nhiều bước như vậy, sẽ nhanh chóng tìm được giá trị cần tìm (không cần phải Full-Scan bảng)

Lợi ích của B-Tree Index

- **Tăng tốc độ truy vấn**: Index B-tree giúp tăng tốc độ truy vấn bằng cách cho phép MySQL nhanh chóng tìm kiếm và truy xuất dữ liệu theo thứ tự sắp xếp.
- **Hỗ trợ sắp xếp**: Index B-tree lưu trữ các giá trị theo thứ tự sắp xếp, giúp các câu lệnh `ORDER BY` và `GROUP BY` trở nên nhanh hơn và hiệu quả hơn.
- **Tăng tốc độ tìm kiếm theo phạm vi**: Index B-tree hỗ trợ tốt cho các câu truy vấn tìm kiếm theo phạm vi (`BETWEEN`, `>`, `<`, `>=`, `<=`), vì các giá trị được lưu trữ theo thứ tự sắp xếp.
- **Giảm I/O**: Index B-tree giảm số lượng truy cập đĩa cần thiết để tìm kiếm dữ liệu, vì MySQL có thể nhảy trực tiếp đến phần dữ liệu cần thiết thay vì duyệt toàn bộ bảng.

Ví dụ sử dụng B-Tree Index

Giả sử bạn có bảng `employees` với các cột `id`, `first_name`, `last_name`, và `hire_date`. Bạn có thể tạo một Index B-tree trên cột `last_name` để tăng tốc độ truy vấn tìm kiếm theo họ:

```
CREATE INDEX idx_last_name ON employees (last_name);
```

Với Index này, truy vấn sau đây sẽ nhanh hơn nhiều vì MySQL có thể sử dụng Index để tìm kiếm nhanh chóng:

```
SELECT * FROM employees WHERE last_name = 'Smith';
```

Tối ưu hóa truy vấn với B-Tree Index

- **Chọn cột để tạo Index**: Tạo index trên các cột mà bạn thường xuyên sử dụng trong các điều kiện WHERE, JOIN, ORDER BY, và GROUP BY.
- **Tránh Index trên các cột có giá trị thay đổi thường xuyên**: Tránh tạo index trên các cột có giá trị thay đổi thường xuyên, vì việc cập nhật index có thể tốn kém tài nguyên.
- **Sử dụng Index đa cột khi cần thiết**: Nếu bạn thường xuyên sử dụng nhiều cột trong các điều kiện tìm kiếm, hãy xem xét tạo index đa cột để tối ưu hóa hiệu suất truy vấn.

Hạn chế của B-Tree Index

- **Không hữu ích cho các truy vấn không bắt đầu từ cột đầu tiên trong Index nhiều cột**: Index B-tree không hữu ích nếu điều kiện tìm kiếm không bắt đầu từ cột đầu tiên của index đa cột.
- **Chi phí cập nhật index**: Việc duy trì Index có thể tốn kém tài nguyên, đặc biệt là khi dữ liệu trong bảng thường xuyên thay đổi.
- **Không hiệu quả cho các cột có ít sự đa dạng (Tính selective thấp)**: Index B-tree không hiệu quả cho các cột có ít sự đa dạng trong giá trị, chẳng hạn như cột chỉ chứa hai hoặc ba giá trị khác nhau.

Hash Indexes

Giới Thiệu

Hash indexes là một loại index trong MySQL sử dụng cấu trúc dữ liệu bảng băm (hash) để lưu trữ và truy xuất các giá trị. Hash indexes đặc biệt hiệu quả cho các truy vấn tìm kiếm chính xác, nơi bạn cần tìm một giá trị cụ thể. Tuy nhiên, chúng không hữu ích cho các truy vấn tìm kiếm phạm vi hoặc sắp xếp dữ liệu.

Cách thức hoạt động của Hash Indexes

Hash indexes hoạt động bằng cách sử dụng một hàm băm để chuyển đổi các giá trị Index thành một số hash, sau đó sử dụng số hash này để tìm kiếm nhanh chóng các bản ghi trong bảng băm. Mỗi số hash trỏ đến một danh sách các bản ghi có cùng số hash.

Ví dụ tạo Hash Indexes

Trong MySQL, hash indexes chủ yếu được hỗ trợ bởi storage engine `MEMORY`. Dưới đây là một ví dụ về cách tạo hash index trên bảng sử dụng storage engine `MEMORY`:

```
CREATE TABLE test (
    id INT PRIMARY KEY,
    name VARCHAR(50),
    INDEX USING HASH (name)
```

```
) ENGINE=MEMORY;
```

Trong ví dụ này, chúng ta tạo một bảng `test` với cột `id` làm khóa chính và cột `name` với hash index.

Lợi ích của Hash Indexes

- **Hiệu suất cao cho tìm kiếm chính xác**: Hash indexes có thể cung cấp hiệu suất cao cho các truy vấn tìm kiếm bằng cách so sánh trực tiếp các giá trị đã băm.
- **Tiết kiệm bộ nhớ**: Hash indexes có thể tiết kiệm bộ nhớ hơn so với B-tree indexes trong một số trường hợp, vì chúng không lưu trữ các giá trị theo thứ tự.

Hạn chế của Hash Indexes

- **Không hỗ trợ tìm kiếm phạm vi**: Hash indexes không thể được sử dụng cho các truy vấn tìm kiếm phạm vi (`<`, `>`, `BETWEEN`) hoặc sắp xếp (`ORDER BY`).
- **Xử lý xung đột Hash**: Khi xảy ra xung đột hash (hai bản ghi khác nhau tạo ra cùng một số hash), hiệu suất của hash indexes có thể bị giảm.
- **Chỉ hỗ trợ một số Storage Engine**: Hash indexes chủ yếu được hỗ trợ bởi storage engine `MEMORY` trong MySQL, do đó, chúng không thể được sử dụng với các storage engine khác như `InnoDB`.

Bonus

Trong InnoDB, Hash Index được triển khai qua tính năng: **Adaptive Hash Index (AHI)**. Adaptive Hash Index trong InnoDB tự động tạo các hash index khi phát hiện rằng một số truy vấn nhất định được thực hiện thường xuyên: nó xây dựng một hash index cho các truy vấn này và lưu trữ trong bộ nhớ (dựa trên kết quả của B-Tree). Điều này giúp B-Tree Index có thêm các khả năng tương tự Hash Index (Ví dụ tìm kiếm các giá trị đã được băm rất nhanh). Quá trình này là hoàn toàn tự động, không cần đến sự can thiệp từ người dùng.

Ví dụ

Giả sử bạn có bảng `users` với các cột `id`, `username`, và `email`, bạn muốn tạo một hash index trên cột `email` để tăng tốc độ tìm kiếm:

```
CREATE TABLE users (
    id INT PRIMARY KEY,
    username VARCHAR(50),
    email VARCHAR(100),
    INDEX email_idx USING HASH (email)
) ENGINE=MEMORY;
```

Với hash index này, truy vấn tìm kiếm sau đây sẽ rất nhanh:

```
SELECT * FROM users WHERE email = 'example@example.com';
```

Full-text Indexes

Full-text indexes là một loại Index đặc biệt trong MySQL, được thiết kế để tìm kiếm từ khóa trong văn bản thay vì so sánh trực tiếp các giá trị trong Index. Full-text search (tìm kiếm toàn văn bản) trong MySQL hữu ích cho các ứng dụng cần thực hiện các truy vấn tìm kiếm phức tạp trong các trường văn bản dài, chẳng hạn như bài viết, bình luận, hoặc mô tả sản phẩm.

Cách thức hoạt động của Full-text Indexes

Full-text indexes hoạt động bằng cách tạo ra một danh sách các từ xuất hiện trong văn bản và lưu trữ vị trí của chúng. Khi một truy vấn tìm kiếm được thực hiện, MySQL sử dụng Index này để tìm kiếm các từ khóa và trả về các kết quả phù hợp. Full-text search có nhiều tính năng phức tạp như từ chặn (stop words), từ gốc (stemming), số nhiều (plurals), và tìm kiếm Boolean.

Tạo Full-text Indexes

Để tạo full-text index trong MySQL, bạn sử dụng từ khóa `FULLTEXT` trong câu lệnh `CREATE TABLE` hoặc `ALTER TABLE`. Ví dụ:

```
CREATE TABLE articles (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    body TEXT,
    FULLTEXT (title, body)
);
```

Hoặc nếu bạn muốn thêm full-text index vào một bảng hiện có:

```
ALTER TABLE articles ADD FULLTEXT (title, body);
```

Truy vấn Full-text Search

Để thực hiện các truy vấn tìm kiếm toàn văn bản, bạn sử dụng câu lệnh `MATCH ... AGAINST`.

Ví dụ:

```
SELECT * FROM articles
WHERE MATCH (title, body) AGAINST ('search keywords');
```

MySQL sẽ trả về các hàng có chứa các từ khóa tìm kiếm trong các cột `title` và `body`.

Các tính năng và kỹ thuật tìm kiếm toàn văn bản

Natural Language Search: Tìm kiếm ngôn ngữ tự nhiên tìm các từ khóa trong văn bản mà không cần sử dụng bất kỳ toán tử so sánh nào. Ví dụ:

```
SELECT * FROM articles WHERE MATCH (title, body) AGAINST ('database indexing');
```

Boolean Mode: Tìm kiếm chế độ Boolean cho phép sử dụng các toán tử như `+`, `-`, `*`, `>` để kiểm soát kết quả tìm kiếm. Ví dụ:

```
SELECT * FROM articles WHERE MATCH (title, body) AGAINST ('+database -indexing' IN BOOLEAN MODE);
```

Query Expansion: Mở rộng truy vấn giúp tìm kiếm các từ liên quan bằng cách sử dụng cú pháp `WITH QUERY EXPANSION`. Ví dụ:

```
SELECT * FROM articles WHERE MATCH (title, body) AGAINST ('database' WITH QUERY EXPANSION);
```

Lợi ích của Full-text Indexes

- **Tìm kiếm nhanh chóng**: Full-text indexes giúp tìm kiếm từ khóa trong văn bản nhanh chóng và hiệu quả.
- **Hỗ trợ tìm kiếm phức tạp**: Hỗ trợ các tìm kiếm phức tạp như tìm kiếm Boolean, tìm kiếm ngôn ngữ tự nhiên, và mở rộng truy vấn.
- **Phù hợp với văn bản dài**: Đặc biệt hữu ích cho các ứng dụng cần tìm kiếm trong các văn bản dài hoặc không có cấu trúc rõ ràng.

Hạn chế của Full-text Indexes

- **Không hỗ trợ tất cả các kiểu cột**: Full-text indexes chỉ hỗ trợ các cột kiểu `CHAR`, `VARCHAR`, và `TEXT`.

- **Không thích hợp cho các truy vấn ngắn**: Không hiệu quả cho các truy vấn rất ngắn hoặc tìm kiếm chính xác.
- **Yêu cầu cấu hình đặc biệt**: Full-text search có thể yêu cầu cấu hình đặc biệt và tối ưu hóa cho hiệu suất tốt nhất.

Ví dụ

Giả sử bạn có bảng `posts` với các cột `id`, `title`, và `content`, bạn muốn tạo một Index toàn văn bản và thực hiện tìm kiếm:

```
CREATE TABLE posts (
    id INT AUTO_INCREMENT PRIMARY KEY,
    title VARCHAR(255),
    content TEXT,
    FULLTEXT (title, content)
);
INSERT INTO posts (title, content) VALUES
('First Post', 'This is the content of the first post'),
('Second Post', 'This is the content of the second post about databases');
-- Tìm kiếm toàn văn bản
SELECT * FROM posts
WHERE MATCH (title, content) AGAINST ('databases');
```

Chiến lược tạo Index hiệu suất cao

Như thế nào là một index tốt

Trong sách **"Relational Database Index Design and the Optimizers",** các tác giả *Tapio Lahdenmaki* và *Mike Leach* giới thiệu một hệ thống đánh giá index với ba sao (three-star index) để giúp đánh giá mức độ phù hợp của index đối với truy vấn. Hệ thống này đánh giá một index theo ba tiêu chí chính:

Các hàng liên quan nằm gần nhau (Adjacent Rows)

Tiêu chí: Index nên sắp xếp các hàng sao cho các hàng có giá trị liên quan nằm gần nhau, giúp giảm thiểu số lượng trang dữ liệu cần đọc.
Ví dụ:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE,
    total DECIMAL(10, 2),
    INDEX (customer_id)
);
```

Trong bảng orders, nếu bạn thường xuyên truy vấn tất cả các đơn hàng của một khách hàng cụ thể, index trên customer_id sẽ giúp các hàng có cùng customer_id nằm gần nhau, giảm thiểu số lượng trang dữ liệu cần đọc.

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

Các hàng được sắp xếp theo thứ tự truy vấn yêu cầu (Sorted Rows)

Tiêu chí: Index nên sắp xếp các hàng theo thứ tự mà truy vấn yêu cầu, giúp giảm thiểu việc sắp xếp dữ liệu tạm thời và tăng hiệu suất.
Ví dụ:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    last_name VARCHAR(50),
    first_name VARCHAR(50),
    hire_date DATE,
    INDEX (last_name, first_name)
);
```

Index trên `last_name` và `first_name` giúp sắp xếp các hàng theo thứ tự trên, hữu ích cho các truy vấn sắp xếp theo tên.

```
SELECT * FROM employees ORDER BY last_name, first_name;
```

Index bao gồm tất cả các cột cần thiết cho truy vấn (Covering Index)

Tiêu chí: Index nên bao gồm tất cả các cột mà truy vấn yêu cầu để MySQL có thể lấy dữ liệu trực tiếp từ index mà không cần truy cập vào bảng chính.

Ví dụ:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100),
    price DECIMAL(10, 2),
    category_id INT,
    INDEX (category_id, price)
);
```

Truy vấn sau có thể được tối ưu hóa bởi index `category_id, price`, vì index bao gồm tất cả các cột cần thiết (`` `category_id` `` và `` `price` ``):

```
SELECT category_id, price FROM products WHERE category_id = 10 ORDER BY price;
```

Chiến lược chọn index

- **Hiểu dữ liệu và truy vấn của bạn**: Trước khi tạo Index, bạn cần hiểu rõ cách dữ liệu của bạn được sử dụng và các loại truy vấn thường xuyên được thực hiện. Điều này giúp bạn xác định các cột cần Index và loại Index phù hợp.
- **Tạo Index cho các cột thường xuyên truy vấn**: Tạo Index trên các cột được sử dụng thường xuyên trong các điều kiện `WHERE`, `JOIN`, `ORDER BY`, và `GROUP BY`. Điều này giúp tăng tốc độ truy vấn và cải thiện hiệu suất.
- **Sử dụng Index đa cột khi cần thiết**: Nếu bạn thường xuyên truy vấn trên nhiều cột, hãy xem xét tạo Index đa cột. Điều này có thể giúp tăng hiệu suất truy vấn so với việc sử dụng nhiều Index đơn cột.
- **Tối ưu hóa thứ tự các cột trong Index đa cột**: Thứ tự của các cột trong Index đa cột rất quan trọng. Đảm bảo rằng các cột được sắp xếp theo thứ tự mà các truy vấn thường xuyên sử dụng nhất. Cột đầu tiên trong Index đa cột phải là cột được lọc nhiều nhất.

Ví dụ về chiến lược Index

Giả sử bạn có bảng `orders` với các cột `order_id`, `customer_id`, `order_date`, và `total_amount`. Bạn thường xuyên truy vấn dữ liệu dựa trên `customer_id` và `order_date`. Bạn có thể tạo index nhiều cột để tối ưu hóa truy vấn này:

```
CREATE INDEX idx_customer_order_date ON orders (customer_id, order_date);
```

Truy vấn sau đây sẽ được tăng tốc độ nhờ index này:

```
SELECT * FROM orders WHERE customer_id = 123 AND order_date = '2024-01-01';
```

Sử dụng Index phủ (Covering Index)

Index phủ là Index chứa tất cả các cột được yêu cầu trong một truy vấn, giúp MySQL lấy dữ liệu trực tiếp từ Index mà không cần truy cập bảng. Điều này giúp giảm I/O và tăng hiệu suất.

Ví dụ, nếu bạn có bảng `employees` và bạn thường xuyên truy vấn các cột `first_name`, `last_name`, và `hire_date`, bạn có thể tạo Index phủ như sau:

```
CREATE INDEX idx_name_hire_date ON employees (first_name, last_name, hire_date);
```

Truy vấn sau đây sẽ được hưởng lợi từ Covering index:

```
SELECT first_name, last_name, hire_date FROM employees WHERE last_name = 'Smith';
```

Index Prefixed (Prefix Indexes)

Đối với các cột văn bản dài, bạn có thể tạo Index trên một phần của giá trị văn bản để tiết kiệm không gian và vẫn đạt hiệu suất tốt. Điều này được gọi là Index prefixed.

Ví dụ, bạn có thể tạo Index trên 10 ký tự đầu tiên của cột `email` trong bảng `users`:

```
CREATE INDEX idx_email_prefix ON users (email(10));
```

Tránh tạo quá nhiều Index

Mặc dù index có thể tăng hiệu suất truy vấn, nhưng chúng cũng làm tăng chi phí lưu trữ và giảm hiệu suất khi thực hiện các thao tác ghi (`INSERT, UPDATE, DELETE`). Do đó, bạn nên tránh tạo quá nhiều index và chỉ tạo index khi cần thiết.

Lựa chọn thứ tự cột tốt

Nguyên tắc lựa chọn thứ tự cột

- **Cột được sử dụng thường xuyên nhất trong điều kiện WHERE**: Đặt các cột được sử dụng thường xuyên nhất trong điều kiện WHERE lên đầu tiên trong index nhiều cột. Điều này giúp MySQL lọc dữ liệu hiệu quả hơn.

- **Ưu tiên cột có độ chọn lọc cao**: Độ chọn lọc của một cột là tỷ lệ số lượng giá trị khác nhau của cột đó so với tổng số hàng. Đặt các cột có độ chọn lọc cao (nhiều giá trị khác nhau) lên trước. Điều này giúp giảm số lượng hàng MySQL cần phải kiểm tra.
- **Cột được sử dụng trong ORDER BY hoặc GROUP BY**: Nếu truy vấn của bạn có sắp xếp (ORDER BY) hoặc nhóm (GROUP BY), đặt các cột này vào thứ tự phù hợp trong Index để MySQL có thể tận dụng Index để sắp xếp hoặc nhóm dữ liệu.
- **Cột được sử dụng trong JOIN**: Nếu truy vấn của bạn có phép nối (JOIN), đặt các cột được sử dụng trong điều kiện JOIN vào Index để tăng tốc độ nối bảng.

Ví dụ

Giả sử bạn có bảng `sales` với các cột `product_id`, `store_id`, `sale_date`, và `amount`. Bạn thường xuyên thực hiện các truy vấn như sau:

```
SELECT * FROM sales
WHERE store_id = 1 AND sale_date BETWEEN '2023-01-01' AND '2023-12-31'
ORDER BY sale_date;
```

Để tối ưu hóa truy vấn này, bạn có thể tạo index đa cột như sau:

```
CREATE INDEX idx_store_date ON sales (store_id, sale_date);
```

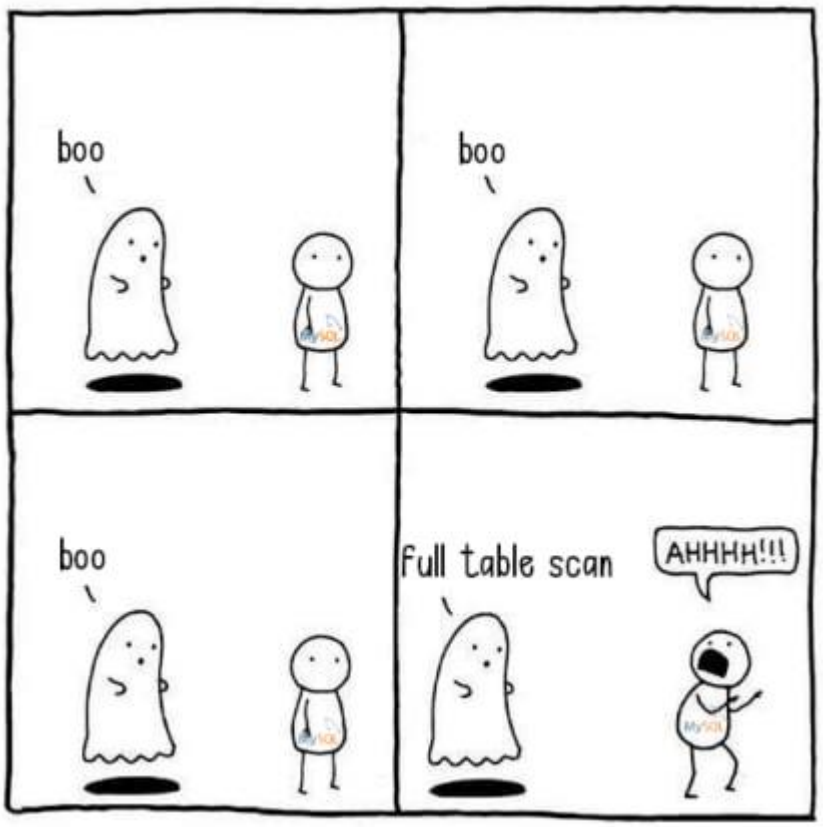
Với index này:

- `store_id` là cột đầu tiên vì nó được sử dụng trong điều kiện WHERE và có độ chọn lọc cao.
- `sale_date` là cột thứ hai vì nó cũng được sử dụng trong điều kiện WHERE và trong ORDER BY.

Lưu ý khi lựa chọn thứ tự cột

Không bỏ qua cột trong Index đa cột: MySQL chỉ có thể sử dụng phần đầu tiên của Index đa cột nếu các cột đó khớp với điều kiện WHERE. Nếu điều kiện WHERE không bao gồm cột đầu tiên trong Index, Index sẽ không được sử dụng hiệu quả.

Một bức ảnh vui về câu chuyện Index và Full table scan 😀 (Index là chiến lược quan trọng để tránh scan toàn bộ bảng)



Tối ưu hóa hiệu suất truy vấn

Tối ưu hóa hiệu suất truy vấn là một trong những kỹ năng quan trọng nhất khi làm việc với cơ sở dữ liệu MySQL. Bằng cách hiểu rõ cách MySQL xử lý truy vấn và áp dụng các kỹ thuật tối ưu hóa, bạn có thể cải thiện đáng kể hiệu suất của ứng dụng. Hãy cùng xem một số cách phổ biến dưới đây nhé:

Sử dụng Index (Indexes)

- **Tạo Index phù hợp**: Tạo Index trên các cột thường xuyên được sử dụng trong các điều kiện WHERE, JOIN, ORDER BY, và GROUP BY.
- **Index phủ (Covering Index)**: Tạo Index chứa tất cả các cột được yêu cầu trong truy vấn để MySQL có thể lấy dữ liệu trực tiếp từ Index mà không cần truy cập bảng.
- **Kiểm tra Index bằng EXPLAIN**: Sử dụng lệnh EXPLAIN để kiểm tra cách MySQL sử dụng index của bạn và điều chỉnh nếu cần.

Tránh sử dụng SELECT *

Chỉ chọn các cột cần thiết: Tránh sử dụng SELECT (*) và chỉ chọn các cột thực sự cần thiết trong truy vấn của bạn. Điều này giúp giảm bớt dữ liệu trả về và tăng tốc độ truy vấn.

```
SELECT first_name, last_name FROM employees WHERE department_id = 5;
```

Tối ưu hóa truy vấn JOIN

- **Sử dụng Index trong JOIN**: Đảm bảo các cột được sử dụng trong điều kiện JOIN có index để tăng tốc độ nối bảng.
- **Hạn chế số lượng JOIN**: Tránh sử dụng quá nhiều JOIN trong một truy vấn nếu có thể. Thay vào đó, xem xét việc chia nhỏ truy vấn thành nhiều truy vấn đơn giản hơn.

Ví dụ một truy vấn JOIN phức tạp:

```
SELECT orders.id, customers.name, products.title FROM orders
JOIN customers ON orders.customer_id = customers.id
JOIN order_items ON orders.id = order_items.order_id
JOIN products ON order_items.product_id = products.id
WHERE customers.country = 'USA';
```

có thể được tách thành nhiều truy vấn đơn giản hơn

```
-- Truy vấn lấy danh sách các đơn hàng từ khách hàng ở USA
SELECT id FROM customers WHERE country = 'USA';

-- Lấy thông tin các đơn hàng
SELECT orders.id, customers.name
FROM orders
JOIN customers ON orders.customer_id = customers.id
WHERE customers.country = 'USA';

-- Lấy chi tiết sản phẩm từ các đơn hàng
SELECT order_items.order_id, products.title
FROM order_items
JOIN products ON order_items.product_id = products.id
WHERE order_items.order_id IN (
    SELECT orders.id
    FROM orders
    JOIN customers ON orders.customer_id = customers.id
    WHERE customers.country = 'USA'
);
```

Sử dụng LIMIT để giới hạn kết quả

Hạn chế số lượng kết quả trả về: Sử dụng LIMIT để giới hạn số lượng rows trả về trong các truy vấn khi không cần thiết phải lấy tất cả dữ liệu.

```
SELECT * FROM orders WHERE customer_id = 123 ORDER BY order_date DESC LIMIT 10;
```

Tối ưu hóa các biểu thức WHERE

Sử dụng biểu thức đơn giản: Tránh sử dụng các biểu thức phức tạp và hàm trong điều kiện WHERE nếu có thể, vì chúng có thể ngăn chặn MySQL sử dụng index.

```
-- Tránh
SELECT * FROM orders WHERE YEAR(order_date) = 2023;
-- Tốt
SELECT * FROM orders WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31';
```

Sử dụng Prepared Statements

Tối ưu hóa truy vấn lặp lại: Sử dụng prepared statements để tối ưu hóa truy vấn lặp lại và giảm chi phí phân tích cú pháp và lập kế hoạch truy vấn. Ngoài ra prepared statements còn giúp bạn phòng tránh SQL Injection và phát hiện sớm các lỗi syntax.

```
PREPARE stmt FROM 'SELECT * FROM orders WHERE customer_id = ? AND order_date =
?';
SET @customer_id = 1;
SET @order_date = '2024-01-01';

EXECUTE stmt USING @customer_id, @order_date;
```

Tối ưu hóa bằng cách sử dụng Caching

Sử dụng Query Cache: Mặc dù MySQL đã loại bỏ query cache trong phiên bản 8.0, bạn có thể sử dụng các hệ thống caching bên ngoài như Redis hoặc Memcached để lưu trữ kết quả truy vấn và giảm tải cho cơ sở dữ liệu.

Sử dụng phân tích và giám sát

- **Phân tích hiệu suất truy vấn**: Sử dụng các công cụ phân tích hiệu suất như MySQL Workbench, pt-query-digest để phân tích và xác định các truy vấn chậm.
- **Giám sát hiệu suất**: Theo dõi các chỉ số hiệu suất của cơ sở dữ liệu như thời gian phản hồi, số lượng kết nối, sử dụng CPU và bộ nhớ để phát hiện và giải quyết các vấn đề kịp thời.

Ví dụ

Giả sử bạn có bảng `transactions` với các cột `id`, `user_id`, `amount`, và `transaction_date`, và bạn muốn tối ưu hóa truy vấn sau:

```
SELECT * FROM transactions WHERE user_id = 123 AND amount > 100 ORDER BY
transaction_date DESC;
```

Các bước tối ưu hóa có thể bao gồm:

Tạo Index đa cột:

```
CREATE INDEX idx_user_amount_date ON transactions (user_id, amount,
transaction_date);
```

Hạn chế kết quả trả về:

```
SELECT * FROM transactions WHERE user_id = 123 AND amount > 100 ORDER BY
transaction_date DESC LIMIT 10;
```

Bonus: Có nên sử dụng SELECT COUNT(*) ?

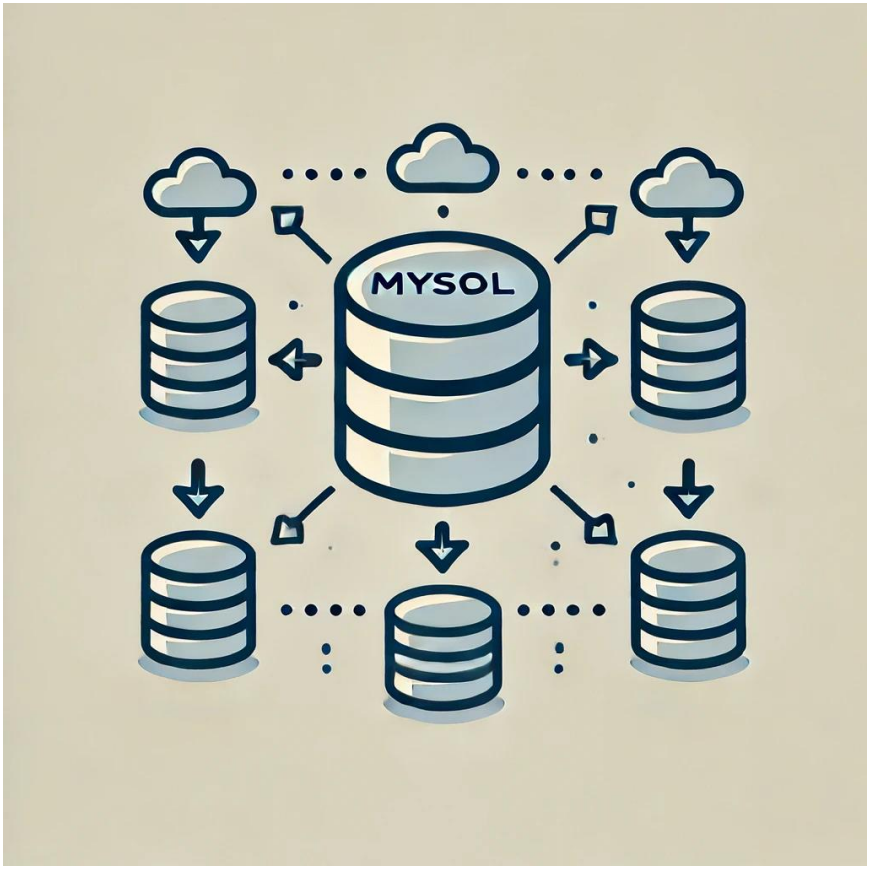
Trong các phần trước, mình có nhấn mạnh rằng bạn không nên sử dụng SELECT(*), mà chỉ nên sử dụng các cột mà bạn cần. Tuy nhiên, điều này liệu có đúng với COUNT(*) ?
Trên thực tế mình đã gặp, hầu hết mọi người, kể cả các senior, cũng đều tránh sử dụng COUNT(*), mà luôn khuyến khích sử dụng COUNT(id). Đây là một quan niệm sai lầm phổ biến về COUNT(): nó không sử dụng toàn bộ các cột trong bảng ! Trong khi SELECT(*) chọn tất cả các cột, COUNT(*) được tối ưu hóa đặc biệt để đếm số hàng. Dấu “*“ ở đây có nghĩa khác nhau trong các ngữ cảnh khác nhau. Trên thực tế SELECT(*) có thể được diễn giải là SELECT([tất cả các cột trong bảng]), còn COUNT(*) sẽ là COUNT([cái nào nhanh nhất]). Vậy COUNT(*) thực sự làm gì?
InnoDB xử lý các câu lệnh COUNT(*) bằng cách tìm kiếm index nhỏ nhất, trừ khi nó nhận được chỉ dẫn trực tiếp từ bạn, hoặc được Optimizer yêu cầu chọn một index khác. Về cơ bản, có thể tóm tắt rằng: COUNT(*) đã được MySQL tối ưu tối đa. Việc thay thế dấu “*” bằng bất cứ cột nào cũng không

làm ảnh hưởng đến nhiều đến performance, vì thế bạn có thể thoải mái sử dụng câu truy vấn này nhé.

Bạn có thể tham khảo thêm tại bài viết sau đây của Percona.

Partition (Phân vùng)

Phân vùng trong MySQL là kỹ thuật chia một bảng lớn thành nhiều phần nhỏ hơn, được gọi là các phân vùng (partitions), để quản lý và truy xuất dữ liệu hiệu quả hơn. Mỗi phân vùng được lưu trữ và quản lý như một bảng riêng biệt nhưng vẫn thuộc cùng một bảng logic.



Bạn có thể tưởng tượng MySQL partition theo mô hình cắt nhỏ như hình

Các loại phân vùng

- **Phân Vùng theo dải (Range Partitioning)**: Dữ liệu được phân chia dựa trên dải giá trị của một cột.
- **Phân Vùng theo danh sách (List Partitioning)**: Dữ liệu được phân chia dựa trên các giá trị cụ thể của một cột.

- **Phân Vùng theo hàm băm (Hash Partitioning)**: Dữ liệu được phân chia dựa trên giá trị hàm băm của một cột.
- **Phân Vùng theo Key (Key Partitioning)**: Tương tự như hash partitioning nhưng sử dụng hàm băm nội bộ của MySQL.

Lợi ích của phân vùng

- **Quản lý dữ liệu dễ dàng hơn**: Dễ dàng quản lý và bảo trì các phân vùng nhỏ hơn so với một bảng lớn.
- **Cải thiện hiệu suất truy vấn**: Giảm bớt số lượng dữ liệu cần quét khi truy vấn, nhờ đó tăng tốc độ truy vấn.
- **Tối ưu hóa xử lý dữ liệu lớn**: Các thao tác như xóa, cập nhật dữ liệu lớn diễn ra nhanh hơn vì chỉ thực hiện trên các phân vùng nhỏ hơn.

Ví dụ về Partition

Phân vùng theo dải (Range Partitioning)

Phân vùng một bảng `sales` theo năm của cột `sale_date`:

```
CREATE TABLE sales (
    sale_id INT,
    amount DECIMAL(10, 2),
    sale_date DATE
)
PARTITION BY RANGE (YEAR(sale_date)) (
    PARTITION p0 VALUES LESS THAN (2020),
    PARTITION p1 VALUES LESS THAN (2021),
    PARTITION p2 VALUES LESS THAN (2022),
    PARTITION p3 VALUES LESS THAN (2023),
    PARTITION p4 VALUES LESS THAN MAXVALUE
```

```
);
```

Phân vùng theo danh sách (List Partitioning)

Phân vùng một bảng `employees` theo cột `department`:

```
CREATE TABLE employees (
    emp_id INT,
    name VARCHAR(50),
    department VARCHAR(50)
)
PARTITION BY LIST COLUMNS(department) (
    PARTITION p0 VALUES IN ('Sales', 'Marketing'),
    PARTITION p1 VALUES IN ('IT', 'Support'),
    PARTITION p2 VALUES IN ('HR', 'Finance')
);
```

Phân vùng theo Hàm Băm (Hash Partitioning)

Phân vùng một bảng `orders` theo cột `order_id`:

```
CREATE TABLE orders (
    order_id INT,
    customer_id INT,
    order_date DATE,
    amount DECIMAL(10, 2)
)
PARTITION BY HASH(order_id) PARTITIONS 4;
```

Quản lý và thao tác với Partition

Thêm Partition:

```
ALTER TABLE sales ADD PARTITION (
    PARTITION p5 VALUES LESS THAN (2024)
);
```

Xóa Partition:

```
ALTER TABLE sales DROP PARTITION p0;
```

Kiểm tra thông tin Partition:

```
SHOW CREATE TABLE sales;
```

Tối ưu hóa hiệu suất truy vấn với Partition

Truy vấn Partition cụ thể: Khi truy vấn dữ liệu, MySQL chỉ quét các phân vùng liên quan, giúp giảm bớt dữ liệu MySQL cần scan, từ đó tối ưu thời gian trả về:

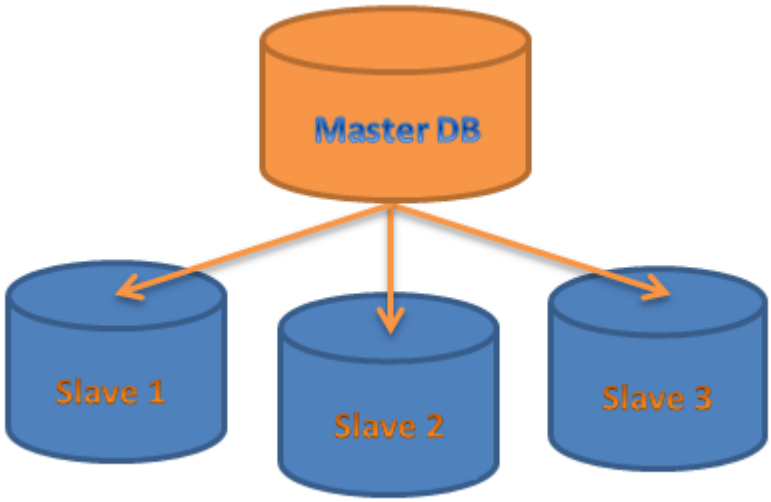
```
SELECT * FROM sales WHERE sale_date BETWEEN '2021-01-01' AND '2021-12-31';
```

Tối ưu hóa các thao tác với partition: Các thao tác như xóa hoặc nén dữ liệu có thể thực hiện trên từng phân vùng riêng biệt, giảm thiểu ảnh hưởng đến toàn bộ bảng.

```
ALTER TABLE sales REORGANIZE PARTITION p1 INTO (
    PARTITION p1a VALUES LESS THAN (2021-06-30),
    PARTITION p1b VALUES LESS THAN (2021-12-31)
);
```

Replication

Replication trong MySQL là một cơ chế cho phép bạn sao chép dữ liệu từ một máy chủ nguồn (master) sang một hoặc nhiều máy chủ sao chép (slave). Điều này không chỉ giúp tăng khả năng mở rộng của hệ thống mà còn cải thiện tính khả dụng và dự phòng cho cơ sở dữ liệu của bạn. Replication là tính năng built-in phổ biến nhất của MySQL khi bạn nghĩ đến việc scale cơ sở dữ liệu của mình.



Hình mô tả đơn giản về tính năng replication trong MySQL

Các loại Replication

- **Statement-Based Replication (SBR)**: Sao chép các câu lệnh SQL thực thi trên máy chủ nguồn sang máy chủ sao chép. Nếu nhìn vào log, bạn sẽ thấy danh sách các câu SQL được thực thi.
- **Row-Based Replication (RBR)**: Sao chép các thay đổi dữ liệu hàng cụ thể từ máy chủ nguồn sang máy chủ sao chép. Nếu nhìn vào log, bạn sẽ thấy dữ liệu trước và sau khi thay đổi của chính xác từng hàng trong CSDL.
- **Mixed-Based Replication (MBR)**: Kết hợp cả hai phương pháp trên, lựa chọn cách sao chép phù hợp nhất cho từng trường hợp cụ thể.

Cách thức hoạt động

Replication hoạt động theo ba bước chính:

- **Ghi nhật ký trên máy chủ master**: Các thay đổi dữ liệu được ghi vào binary log dưới dạng các sự kiện (events).
- **Sao chép nhật ký**: Các Slave (Máy chủ phụ) sao chép các sự kiện từ binary log của máy chủ nguồn về relay log của mình.
- **Phát lại các sự kiện**: Slave chạy lại các sự kiện từ relay log để áp dụng các thay đổi vào cơ sở dữ liệu của mình.

Các tình huống sử dụng Replication

- **Phân tán dữ liệu**: Sử dụng replication để sao chép dữ liệu đến các trung tâm dữ liệu khác nhau để đảm bảo tính sẵn sàng cao và giảm độ trễ.
- **Tăng cường khả năng đọc**: Phân phối các truy vấn đọc đến các slaves để giảm tải cho master.
- **Dự phòng và khôi phục**: Sử dụng slave làm nguồn dự phòng để nhanh chóng khôi phục hoạt động nếu máy chủ master gặp sự cố.
- **Phân tích dữ liệu**: Chạy các truy vấn phân tích nặng trên slave để tránh ảnh hưởng đến hiệu suất của master.

Lời kết

Bạn đã đọc hết cuốn “MySQL cho người đi làm - Các chiến lược tối ưu MySQL cơ bản” của mình. Từ đáy lòng, mình cám ơn bạn rất nhiều vì đã đọc đến cuối quyển Ebook này. Có thể cách viết mình còn lộn xộn, mình mong bạn hiểu những tâm huyết của mình, và những mong mỏi được chia sẻ kiến thức đến cộng đồng. Làm việc và tối ưu cơ sở dữ liệu (đặc biệt với MySQL) là một trong những kiến thức nền tảng cực kỳ quan trọng, cho dù bạn làm với techstack nào. Mình hy vọng ebook này sẽ giúp đỡ bạn phần nào trên con đường học lập trình nhé 😀.

Cám ơn bạn rất nhiều !

Nếu bạn thích những gì trong cuốn sách này và muốn trao đổi thêm, đừng ngần ngại liên lạc với mình qua:

Email: huynt57@gmail.com

Facebook: Tại đây

Cuốn Ebook này sẽ không tồn tại nếu không có vợ mình Dương Thị Thu Huyền và con trai mình, cố vấn tí hon Nguyễn Dương Hoàng Khôi. Cám ơn gia đình đã luôn bên cạnh và ủng hộ mình.

Happy Coding !